Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku

Snježana Mijošević

Baza podataka za sveučilišnu društvenu mrežu

Diplomski rad

Osijek, *2014.*

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku

Snježana Mijošević

Baza podataka za sveučilišnu društvenu mrežu

Diplomski rad

Mentor: izv. prof. dr. sc. Domagoj Matijević
Komentor: Slobodan Jelić

Osijek, *2014.*

Sadržaj

1 Uvod

U današnjem svijetu tematika društvenih mreža je jako aktualna tematika među mladima, budući da su upravo mlade osobe oni koji najviše vremena provode na njima. Osvrnimo se najprije na povijesni aspekt društvenih mreža, potom spomenimo činjenicu kako je *Facebook* u početku služio kao društvena mreža na sveučilištu, da bismo naposljetku objasnili kao izbor teme ovog diplomskog rada upravo bazu podataka za sveučilišnu društvenu mrežu.

Povijest društvenih mreža vezana je uz tzv. *Usenet* - preteču današnjih foruma - 90-ih godina 20. stoljeća. Zapravo, forumi se smatraju prvim društvenim mrežama u svijetu. Iako su nudili opciju komentiranja različitih tema, debate ili virtualnog druženja, nedostajala im je najvažnija opcija većine današnjih društvenih mreža - mogućnost postati s nekim virtualni prijatelj. Osim toga, u forumima se u početku registriralo pseudonimom. Prva prava društvena mreža sa svim funkcijama koje imaju današnje društvene mreže je bila društvena mreža *SixDegrees*. Svatko se mogao registrirati na stranicu (obvezno imenom i prezimenom) i pozvati e-mailom prijatelje, te pretraživati korisnike po njihovim zanimanjima ili interesima. *SixDegrees* je službeno propao 2001. godine. S vremenom su nastajale društvene mreže specijalizirane za razna područja i interese, među kojima je bio poznat *Friendster* (nastao u veljači 2003. godine) - društvena mreža za komunikaciju s prijateljima, a potom se pojavio *LinkedIn* (poslovna društvena mreža) i *MySpace* (osnovan u kolovozu 2003. godine - dopuštao sve što korisnik želi: lažno ime, blog, igre, otvaranje profila bez pozivnice postojećeg člana).

Prva društvena mreža koja je bila specijalizirana za studente sveučilišta je *Facebook*, kojeg je pokrenuo Mark Zuckerberg, student s Hardvarda zajedno sa svojim cimerima, kako bi fotoalbume studenata postavio online, gdje bi ih svi mogli vidjeti. Ispočetka je služio samo fakultetima, a potom je 2006. godine otvoren za sve ljude. 2007. godine je Zuckerberg omogućio svakom programeru da osmisli i postavi aplikaciju na stranicu. U današnje vrijeme raste popularnost nove društvene mreže *Google+*, koja omogućuje grupiranje korisnika s kojima dijelimo informacije u krugove (npr. obitelj, prijatelji, poznanici,...), te dijeljenje određenih informacija samo s izabranim krugovima ljudi, a kako je za logiranje potreban samo račun na *Google*-u, ta bi mreža mogla predstavljati konkurenciju *Facebook*-u.

U ovom je radu izrađena baza podataka za sveučilišnu društvenu mrežu. Zapravo, radi se o dvije baze podataka koje imaju jednaku relacijsku shemu. Jedna je aktivna (u njoj se sve odvija), a druga povijesna (iz nje se ništa ne briše). Za razliku od *Facebook*-a, tj. od onog što je *Facebook* s vremenom postao, naša društvena mreža ima korisnike koji su isključivo

vezani uz sveučilište: profesore, studente i gostujuće studente, te sadrži informacije o ustanovama obrazovanja, profesorima na pojedinim fakultetima, kabinetima,... pa baza osim podataka vezanih uz korisnike, njihove osobne informacije, poruke, slike i sl. sadrži i podatke o obrazovanju, generacijama obrazovanja u pojedinoj ustanovi, kolegijima koji su upisani na određenom smjeru i sl., odnosno podatke vezane uz sveučilište i akademsko obrazovanje.

U radu je najprije naveden logički dizajn baze podataka (kriteriji za bazu i normalizacija), a potom implementacija baze podataka, s naglaskom na gradivo vezano uz ograničenja, okidače, prikaze, indekse, pohranjene procedure i funkcije. U Dodacima se nalazi relacijska shema baze podataka, te objašnjenje priloga koji su na CD-u priloženom uz rad.

2 Logički dizajn baze podataka

U ovom ćemo dijelu rada konkretnije progovoriti o relacijskom modelu podataka, o relacijama, te o funkcijskim ovisnostima.

Spomenimo prvo najbitnije pojmove. Relacijski model podataka prikazuje podatke pomoću dvodimenzionalnih tablica ili *relacija*. *Instancom relacije* nazivamo tablicu kojom je predstavljen skup entiteta u relacijskom modelu podataka, a *relacijskom shemom* nazivamo ime relacije `R` iza kojeg slijedi popis njezinih atributa u zagradi, dok je *relacijska shema baze podataka* skup relacijskih shema svih relacija u nekoj bazi podataka. Navedimo sada najprije tvrdnje vezane uz našu bazu podataka za sveučilišnu društvenu mrežu, a potom ćemo navesti relacijsku shemu.

2.1 Kriteriji za model baze

Kao kriteriji za model baze, poslužit će nam sljedeće tvrdnje, a relacijska shema baze podataka nalazi se u Dodatku A.

- Svaki korisnik u bazi podataka ima naveden svoj korisnički broj (`userID`), korisničko ime (`username`), šifru (`password`), `ime`, `prezime`, `spol`, datum rođenja (`datum_rodjenja`), i mjesto rođenja (`mjesto_rodjenja`). Pri tome su korisnički broj i korisničko ime jedinstveni, a šifra i korisničko ime mogu imati najviše 10 znakova.

- U bazi postoji popis gradova i država. Za svaki grad je naveden `gradID`, poštanski broj[1] (`postanski_broj`), naziv grada (`naziv_grada`) i država u kojoj se nalazi.

- Među `korisnike` se ubrajaju `profesori` (pod tim se misli ne samo na one koji imaju titulu profesora, nego i na docente, asistente,...), `studenti`[2] (s promatranog sveučilišta) i `gostujući studenti` (s drugih sveučilišta i ustanova koje nisu na promatranom sveučilištu). Odnosno, `userID` može biti `profesorID`, `sID`, `gID` ili neka kombinacija navedenih atributa.

- Svakom `studentu` navedena je `adresa` (pri tome u bazi postoji popis `ulica`, svakoj je pridružen `grad`, a preko grada i država u kojoj se nalazi), broj telefona (`telefon`), broj mobitela (`mobitel`) i e-mail adresa (`email`).

[1] Pri tome za neke gradove iz drugih država, za koje ćemo imati gostujuće studente u bazi i koji imaju više poštanskih brojeva, uzimamo onaj poštanski broj koji odgovara području grada na kojem se ustanova ili student nalazi. No, takvih primjera nećemo imati puno, pa tu činjenicu možemo smatrati zanemarivom u bazi.

[2] Pod tim pojmom u bazi ćemo misliti isključivo na one studente koji su s promatranog sveučilišta, tako da to ubuduće neće biti posebno naglašavano. Bude li trebalo govoriti o studentima s drugih sveučilišta, koristit ćemo pojam gostujućeg studenta.

- Svakom `profesoru` je navedena titula (`nastavna` i `znanstvena`), a može biti navedena `adresa`, `telefon`, `mobitel` i e-mail adresa (`email`) - sve privatno, ali ne mora.
- Svaki `gostujući student` ima navedenu `ustanovu obrazovanja` s koje dolazi, `jezike` koje govori (u bazi imamo popis jezika), potom `znanstveno-istraživačka područja` koja ga zanimaju (u bazi imamo popis znanstveno-istraživačkih područja) i koje `kolegije` sluša na kojem smjeru (s popisa izvođenja kolegija u bazi na našem sveučilištu).
- Svaki `korisnik` može (ali i ne mora) popuniti osobne podatke - čime se bavi (`aktivnost`), što ga zanima (`interes`), koju glazbu sluša (`glazba`), koje TV emisije i serije voli (`TV_emisije_i_serije`), koje filmove voli (`filmovi`), koje su mu knjige najdraže (`knjige`), i što bi sam izrekao o sebi (`o_sebi`).
- Omogućeno je slanje `poruka` među `korisnicima`, svakoj je zabilježen ID pošiljatelja (`posiljateljID`), ID primatelja (`primateljID`), vrijeme slanja (`vrijeme_slanja`) i `tekst`.
- Omogućen je `spomenar`. Svaki `korisnik` može svakome napisati nešto za uspomenu, a zabilježeni su korisnički broj osobe koja piše (`tko_pise`), korisnički broj osobe čiji je spomenar u pitanju (`kome_pise`), poruka i vrijeme objave[3]. Pri tome vrijedi, ako se osoba koja je nekome zapisala poruku u spomenar obriše iz mreže, poruka ostaje, ali se atribut `tko_pise` postavlja na `NULL`.
- U bazi postoji popis ustanova obrazovanja. Za svaku je naveden broj ustanove (`ustanovaID`), naziv ustanove (`naziv_ustanove`), `adresa`, broj telefona (`telefon`) i `fax`. Pri tome, ako se neka ustanova nalazi na više lokacija, za svaku je navedena `adresa`, `telefon` i `fax`.
- U bazi postoji popis vrsta obrazovanja u pojedinoj ustanovi, i o generacijama svake vrste obrazovanja (godina početka i godina završetka).
- Svaki `korisnik` može imati navedene podatke o svom obrazovanju.
- Svakom `korisniku` je dopušteno učitavanje `slika`, te zapis informacija `o pojedinoj slici`.
- Svaki `fakultet` ima navedenu svoju `web-stranicu` i `dekana` (koji je ujedno i `profesor`).

[3] Za ime atributa stoji riječ `datum` tipa `TIMESTAMP` jer smatramo kako bi slučajevi zapisa iste osobe istoj osobi dva ili više puta dnevno bili iznimno rijetki, ali su dopušteni na ovaj način (ne mora značiti da će svatko unijeti sat u kojem je pisao, a osigurana je jedinstvenost zapisa za osobu koja piše, osobu čiji je spomenar, tekst poruke i vrijeme).

- Za svaki `fakultet` je navedeno koji `smjerovi` na njemu postoje, a za svaki `smjer` je naveden `tip studija` i `trajanje` u semestrima.
- Svaki `profesor` na svakom `fakultetu` na kojem radi može imati navedenu `e-mail adresu`, a mora imati naveden `postotni dio radnog vremena`.
- U bazi je naveden popis `kabineta` na pojedinim `fakultetima`, za svaki kabinet je navedena `oznaka`, `telefon` i `fax`. U bazi ćemo pretpostaviti da svaki kabinet ima samo jedan broj telefona i samo jedan broj faxa.
- Svaki `profesor` na svakom `fakultetu` na kojem radi može imati svoj `kabinet`. Pri tome, nije dozvoljeno da profesor ima kabinet na fakultetu na kojem ne radi.
- Dozvoljeno je da svaki `profesor` na svakom `fakultetu` na kojem radi ima svoju `web-stranicu`.
- `Profesor` može držati `konzultacije` na onima `fakultetima` na kojima radi.
- U bazi postoji popis `znanstveno-istraživačkih područja`. Svaki `profesor` se može baviti s više `znanstveno-istraživačkih područja`, i svako `znanstveno-istraživačko područje` može imati više `profesora` koji se njim bave.
- Svako `znanstveno-istraživačko područje` može imati `grupu profesora` koja se njime bavi. Svakoj `grupi profesora` naveden je `naziv` i `opis`. Pri tome, u grupi profesora ne može biti nijedan profesor koji se ne bavi znanstveno-istraživačkim područjem za koje je grupa namijenjena.
- U svakoj `grupi profesora`, članovi mogu stavljati `objave`[4] i `komentare`. Pri tome se bilježi `vrijeme objave`. Ostale osobe na mreži, koje nisu članovi grupe, ne mogu ni vidjeti[5] `objave` i `komentare` u njoj.
- Za pojedinog `studenta` je navedeno na kojim `smjerovima` studira (jedan `student` može studirati na više `smjerova`), a za svaki smjer na kojem studira naveden je `semestar`, `broj indeksa` i `status` (npr. je li redoviti student, izvanredni ili s plaćanjem).

[4] engl. post

[5] U bazi je s tehničke strane (preko okidača) osigurano eliminiranje komentara na objave grupe sa strane korisnika koji grupi ne pripadaju. Inače se takve stvari riješavaju na razini aplikacije, a ne na razini baze, tako da su okidači koji brišu krive unose u bazi postavljeni više zbog primjera korištenja okidača, nego zbog rješavanja problema na najefikasniji način. Budući je ovaj rad usmjeren isključivo na razvoj baze podataka za sveučilišnu društvenu mrežu ne baveći se pri tome aplikacijskom razinom, okidače koristimo za rješavanje pojedinih problema na razini baze podataka.

- U bazi postoji popis `kolegija`. Svaki `kolegij` može imati potrebna `predznanja` (također `kolegije`) na svakom `smjeru` na kojem `se izvodi`. Pri tome, neki smjerovi mogu biti u planiranju, tj. ne mora se svaki kolegij za koji su navedena predznanja na nekom smjeru izvoditi.
- Svaki `kolegij` na svakom `smjeru` na kojem `se izvodi` može imati navedenu `web-stranicu`, a mora imati naveden `semestar` na kojem se izvodi i `ECTS bodove`.
- Pojedini `kolegij` na pojedinom `smjeru` može imati više `profesora` koji rade na njemu.
- U bazi postoji popis kolegija koje `student` trenutno sluša na pojedinom smjeru.
- `Studenti` pojedinog `smjera` i `godine`, ili samo pojedinog `smjera`, mogu se upisati u `grupu` za to predviđenu, te slično kao i profesori u grupama profesora mogu stavljati `objave` i `komentare`. Pri tome, studenti koji nisu s tog smjera i te godine, ili samo tog smjera (ovisno o specifikaciji grupe), ne mogu biti članovi grupe i ne mogu objavljivati[6] postove i komentare.
- Pri objavi `postova` i `komentara` u `grupama studenata`, bilježi se `tekst` i `vrijeme objave`.
- Svaki `korisnik` može izabrati s kojim `korisnicima` će podijeliti svoje `osobne podatke` i `spomenar`.
- Ako obrišemo `profesora` iz `grupe profesora` ili `studenta` iz `grupe studenata`, obrisat će se i sve njihove `objave` i svi njihovi `komentari` (ubrzo nakon toga).

2.2 Normalizacija baze podataka

U svrhu normalizacije baze podataka, uvodimo pojam funkcijske ovisnosti.

Definicija 2.1 (Funkcijska ovisnost) *Za relaciju* `R(A, B)`*, skup atributa* `B` *je* ***funkcijski ovisan*** *o skupu atributa* `A` *i pišemo* `A` $\rightarrow$ `B` *ako za svaku vrijednost od* `A` *postoji samo jedna vrijednost* `B`*, odnosno ako vrijedi da za dvije* n*-torke s jednakom vrijednošću* `A` *koje u isto vrijeme postoje u relaciji, ta činjenica ujedno znači da te dvije* n*-torke imaju jednaku vrijednost* `B`*.*

Definicija 2.2 (Potpuna funkcijska ovisnost) *Za relaciju* `R(A, B)`*, skup atributa* `B` *je* ***potpuno funkcijski ovisan*** *o skupu atributa* `A`*, ako vrijedi da je* `B` *funkcijski ovisan o* `A` (`A` $\rightarrow$ `B`)*, ali* `B` *nije funkcijski ovisan ni o jednom pravom podskupu od* `A`*.*

[6] Na analogan način kao i u prethodnoj fusnoti, problem koji bi se najefikasnije riješio na razini aplikacije, u bazi je riješen preko okidača. (Primjerice, netko u tablicu za upis korisnika u grupu želi unijeti onog korisnika koji toj grupi ne bi trebao pripadati, pa je osigurano brisanje takvih redaka i sličnih slučajeva.)

Definicija 2.3 (Ključ relacije) *Za relaciju* `R(A, B)`*, u kojoj su* `A` *i* `B` *dva disjunktna skupa atributa, skup atributa* `A` *je* ***ključ relacije*** *ako je skup atributa* `B` *potpuno funkcijski ovisan o njemu.*

Propozicija 2.1 (Pravilo kombiniranja) *Ako imamo skup atributa* $\{A_1,\ A_2, \ldots, A_n\}$ *i ako on funkcijski povlači svaki pojedini element iz skupa atributa* $\{B_1,\ B_2, \ldots, B_m\}$*, tada skup atributa* $\{A_1,\ A_2, \ldots, A_n\}$ *funkcijski povlači skup* $\{B_1,\ B_2, \ldots, B_m\}$*. Odnosno,*

$A_1\ A_2 \ldots A_n \rightarrow B_1$
$A_1\ A_2 \ldots A_n \rightarrow B_2$
$\ldots$
$A_1\ A_2 \ldots A_n \rightarrow B_m$

je ekvivalentno sljedećoj funkcijskoj ovisnosti:

$A_1\ A_2 \ldots A_n \rightarrow B_1\ B_2 \ldots B_m$.

Pogledajmo sada primjere funkcijskih ovisnosti atributa u relacijama `korisnik` i `poruke`.

Primjer 2.1 (Funkcijske ovisnosti i ključevi relacije `korisnik`) *Svrha ovog primjera je pokazati neke funkcijske ovisnosti relacije* `korisnik`*, iz kojih slijedi da su atributi* `userID` *i* `username` *ključevi. Navedimo sada ponovno relacijsku shemu relacije* `korisnik`*:*

`korisnik(userID, username, password, ime, prezime, spol, datum_rodjenja, mjesto_rodjenja)`

Kako u relaciji `korisnik` *imamo atribut* `userID` *koji jednoznačno određuje svaki drugi atribut, vrijede sljedeće funkcijske ovisnosti:*

`userID` $\rightarrow$ `username`
`userID` $\rightarrow$ `password`
`userID` $\rightarrow$ `ime`
`userID` $\rightarrow$ `prezime`
`userID` $\rightarrow$ `spol`
`userID` $\rightarrow$ `datum_rodjenja`
`userID` $\rightarrow$ `mjesto_rodjenja`

Kako vrijedi pravilo kombiniranja, tada nam vrijedi i sljedeća funkcijska ovisnost:

`userID` $\rightarrow$ `username password ime prezime spol datum_rodjenja mjesto_rodjenja`

U relaciji `korisnik` *također postoji još jedan atribut koji je jedinstven - atribut* `username`*. Stoga vrijede sljedeće funkcijske ovisnosti:*

`username` → `userID`
`username` → `password`
`username` → `ime`
`username` → `prezime`
`username` → `spol`
`username` → `datum_rodjenja`
`username` → `mjesto_rodjenja`

Ovdje bismo mogli dodati sljedeću funkcijsku ovisnost koje dobijemo zbog pravila kombiniranja:

`username` → `userID password ime prezime spol datum_rodjenja mjesto_rodjenja`

Sada iz Definicije 2.3 *vidimo da su atributi* `userID` *i* `username` *ključevi relacije* `korisnik`.

Primjer 2.2 (Funkcijske ovisnosti atributa relacije `poruke`) *Promotrimo relaciju* `poruke`*, s relacijskom shemom:*

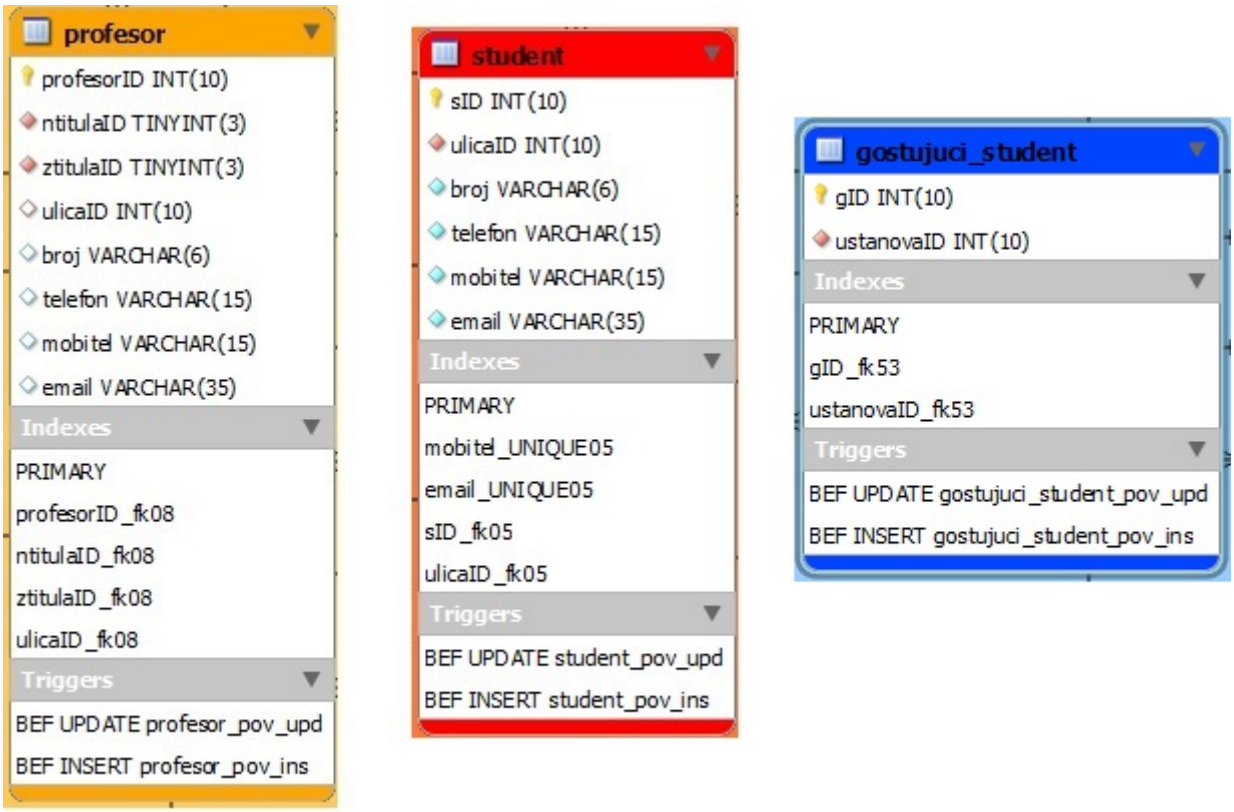
`poruke(posiljateljID, primateljID, vrijeme_slanja, tekstID)`

Jedan pošiljatelj može poslati poruku više primatelja u isto vrijeme. Jedan primatelj može dobiti poruku od više pošiljatelja u isto vrijeme (teoretski moguće, ali rijetko u praksi). Jedan pošiljatelj može jednom primatelju poslati više poruka (svaku u drugo vrijeme), stoga će u toj relaciji vrijediti sljedeća funkcijska ovisnost:

`posiljateljID primateljID vrijeme_slanja` → `tekstID`.

Proces kojim se iz baze, odnosno iz relacijskih shema tablica u bazi uklanjaju parcijalne, tranzitivne i višeznačne ovisnosti, naziva se normalizacija.

Definicija 2.4 (Prva normalna forma) *Kažemo da je relacija u* ***prvoj normalnoj formi (1NF)****, ako je vrijednost svakog atributa nedjeljiva (jednostruka) - tj. kad je vrijednost svakog atributa samo jedna vrijednost iz domene tog atributa, te kad neključni atributi relacije funkcijski ovise o ključu relacije.*



Slika 2.1 *Relacijske sheme relacija* `profesor`, `student` *i* `gostujuci_student`

Promotrit ćemo sada samo one relacije koje nam mogu biti sporne po pitanju 1NF. Najprije pogledajmo relaciju `student` i relaciju `profesor`. U njima se traži samo jedan broj mobitela, jedan broj telefona i jedna e-mail adresa, a tako se i pretpostavlja da će kao korisnici unijeti. Ukoliko netko ima više brojeva mobitela, telefona, ili više e-mail adresa, upisuje samo onu primarnu. Tipovi podatka su tako osmišljeni da ne dozvoljavaju unos dvije vrijednosti za atribut (`telefon VARCHAR(15)`, `mobitel VARCHAR(15)`, `email VARCHAR(35)`), a na aplikacijskoj razini, kad bi se baš išlo napraviti web sučelje za bazu, moglo bi se preko regularnih izraza spriječiti čak i pokušaj unosa dvostruke vrijednosti tako da se onemogući razmak.

Promotrimo sada relaciju `gostujuci_student`. Iako student može studirati na više fakulteta, ako se radi o gostujućem studentu, on dolazi u ime samo jedne ustanove na kojoj studira, pa je relacija u 1NF. Kako su i sve ostale relacije (osim nabrojanih) u 1NF, baza je u 1NF.

Definicija 2.5 (Druga normalna forma) *Relacija je u* ***drugoj normalnoj formi (2NF)****, ako je u 1NF i ako je svaki nekljčni atribut potpuno funkcijski ovisan o bilo kojem ključu relacije.*

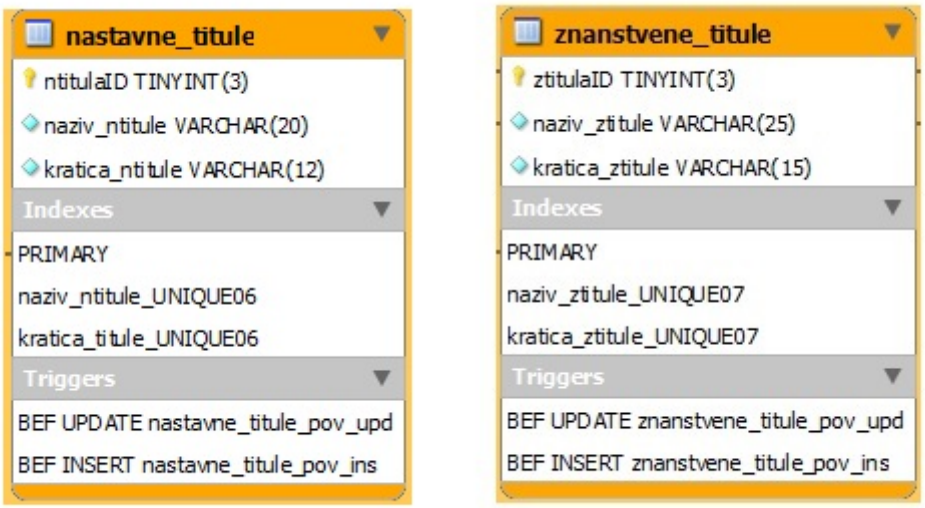
U bazi je u svakoj relaciji svaki neključni atribut potpuno funkcijski ovisan o bilo kojem ključu relacije, tj. sve relacije su u 2NF, pa je i baza u 2NF.

Da bismo govorili o trećoj normalnoj formi, najprije moramo definirati pojam tranzitivne ovisnosti.

Definicija 2.6 (Tranzitivna ovisnost) *Zadana je relacijska shema* `R(X, Y, Z)` *i skupovi atributa* `X` $\subseteq$ `R`, `Y` $\subseteq$ `R`, `Z` $\subseteq$ `R`. *Skup atributa* `Z` *je* ***tranzitivno ovisan*** *o skupu atributa* `X` *ako vrijedi* `X` $\rightarrow$ `Y`, `Y` *ne povlači funkcijski* `X` *i* `Y` $\rightarrow$ `Z`.

Sada možemo definirati i treću normalnu formu.

Definicija 2.7 (Treća normalna forma) *Relacijska shema je u* ***trećoj normalnoj formi (3NF)*** *ako je u 1NF i ako nijedan atribut iz zavisnog dijela (tj. atribut s desne strane strelice u funkcijskoj ovisnosti) nije tranzitivno funkcijski ovisan o bilo kojem ključu relacije.*



Slika 2.2 *Relacijske sheme relacija* `nastavne_titule` *i* `znanstvene_titule`

Promotrimo sada relacije koje nam na prvi pogled mogu djelovati sporne po pitanju tranzitivne ovisnosti. Radi se o relacijama `nastavne_titule` i `znanstvene_titule`. Budući da imaju prilično slične relacijske sheme, dovoljno je na jednoj objasniti slučaj. Uzmimo za primjer relaciju `nastavne_titule` i njezine atribute. Promotrimo funkcijske ovisnosti koje u njoj vrijede:

`ntitulaID` $\rightarrow$ `naziv_ntitule`
`naziv_ntitule` $\rightarrow$ `kratica_ntitule`

`naziv_ntitule` $\rightarrow$ `ntitulaID`.

Uočimo, zbog zadnje funkcijske ovisnosti se ipak ne radi o tranzitivnoj ovisnosti atributa `ntitulaID` i atributa `kratica_ntitule`, pa je relacija u 3NF. Na isti način se pokaže za relaciju `znanstvene_titule` i njezine atribute.

Prema [4, str. 39], navedimo najprije definiciju natključa relacije, a potom možemo definirati Boyce-Coddovu normalnu formu.

Definicija 2.8 (Natključ relacije) *Za* k*-člani skup atributa* $\{A_{i_1} \dots A_{i_k}\} \subseteq \{A_1 \dots A_n\}$ *kažemo da je* ***natključ relacije*** `R` *(koja ima atribute* $A_1 \dots A_n$*),* $1 \leq k \leq n$*, ako on funkcijski određuje sve atribute iz pripadne relacijske sheme.*

Definicija 2.9 (Boyce-Coddova normalna forma) *Kažemo da je relacija* `R` *u* ***Boyce-Coddovoj normalnoj formi (BCNF)*** *ako u svakoj netrivijalnoj*[7] *funkcijskoj ovisnosti oblika* $A_1 \dots A_n \rightarrow B_1 \dots B_m$*, skup atributa* $\{A_1, \dots A_n\}$ *predstavlja natključ relacije* `R`*, odnosno ako je u 1NF i ako niti jedan atribut nije tranzitivno ovisan o bilo kojem ključu relacije.*

Promotrimo ponovno funkcijske ovisnosti iz *Primjera 2.1.* Uočimo kako atribut `userID` funkcijski određuje sve atribute iz pripadne relacijske sheme, pa je on natključ relacije (`korisnik`). Također, atribut `username` funkcijski određuje sve atribute pripadne relacijske sheme, pa je i on natključ iste relacije. Svaka tablica u bazi je u BCNF, pa je i cijela baza u BCNF.

U praksi se najčešće baza dovodi do 3NF i rijetki su slučajevi (iako postoje) da baza podataka zadovoljava 3NF, a istovremeno ne zadovoljava BCNF. Zato ćemo kod toga stati s normalizacijom. Postoje još 4NF, 5NF i 6NF (vidi [5]).

[7] Netrivijalna funkcijska ovisnost je ona funkcijska ovisnost u kojoj skup atributa s lijeve strane strelice ili uopće ne povlači sam sebe ili neki od svojih podskupova, ili uz to povlači bar još jedan atribut koji nije s lijeve strane strelice.

3 Implementacija baze podataka

Baza podataka za sveučilišnu društvenu mrežu implementirana je u MySQL-u, najprije u verziji 5.6.15, a potom u verziji 5.7.3-m13. Grafički prikaz baze podataka izrađen je preko alata MySQL Workbench. Detaljnije o grafičkom prikazu i o implementaciji nalazi se u Dodatku B.

U ovom poglavlju govorit ćemo o ograničenjima, okidačima, prikazima, indeksima, te o pohranjenim procedurama i funkcijama.

3.1 Ograničenja i okidači

Kreirajući bazu podataka, suočavamo se i s mogućnošću da promjena, brisanje ili unos novih podataka u bazu mogu biti pogrešni na nekoliko načina. Jedan od primjera mogu biti pogreške u prijepisu pri ručnom unosu podataka. Zbog toga će nam biti potrebna ograničenja[8] i okidači[9], tzv. *aktivni* elementi SQL-a. Radi se o izrazima ili tvrdnjama koje jednom spremimo u bazu, očekujući da će se izvršiti u pravi trenutak (npr. prije ili poslije unosa novog retka, brisanja starog ili modifikacije postojećeg). Pri tome, ograničenja koristimo kako bismo dohvatili pogreške pri unosu podataka, dok su okidači dinamičniji koncept. Pojasnimo to na ovaj način: dok ograničenja govore o stanju baze, okidači govore o tome na koji se način baza razvija dalje.

Okidači su poznati po tome što reagiraju na neki događaj (npr. unos novog retka, brisanje starog ili modifikacija postojećeg), te provjeravaju uvjete na bazi, i u slučaju da su uvjeti ispunjeni, izvršavaju akciju (ili akcije). Uvjet može biti prekršaj ograničenja ili nešto općenitije.

Dakle, ograničenja i okidači služe kako bismo nadgledali stanje u bazi.

3.1.1 Ograničenja

Navedimo sada vrste ograničenja kojima ćemo se detaljnije baviti u MySQL standardu:

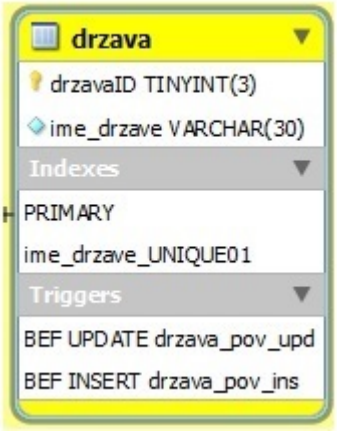
- ograničenja ne `NULL` vrijednosti[10],

[8] engl. constraints
[9] engl. triggers
[10] engl. not-`NULL` constraints

- ograničenja na ključ[11] i strani ključ[12] (referencijalni integritet[13]).

Ograničenja ne NULL vrijednosti

Prilikom susreta s bazama podataka i tablicama, nerijetko se događa slučaj kada želimo obavezan unos nekog atributa u n-torci, tj. ne želimo dopustiti mogućnost da takav atribut poprimi `NULL` vrijednost. To ograničenje se postiže stavljanjem riječi `NOT NULL` nakon tipa podatka pojedinog atributa pri definiranju tablice.



Slika 3.1 *Relacijska shema relacije* `drzava`

Primjer 3.1 (Primjer `NOT NULL` vrijednosti atributa) *Promotrimo tablicu* `drzava` *(njezina relacijska shema se nalazi na* Slici 3.1*). Ne bi bilo praktično kada bismo dopustili da bilo koji od atributa* `drzavaID` *i* `ime_drzave` *ima vrijednost* `NULL`*. Zbog toga u kodu stvaranja tablice stavljamo pored tipa podatka svakog od ta dva atributa ključne riječi* `NOT NULL`*:*

```
CREATE TABLE IF NOT EXISTS drzava (
  drzavaID TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
  ime_drzave VARCHAR(30) NOT NULL,
  PRIMARY KEY (drzavaID))
ENGINE = InnoDB;
```

Ograničenja na ključ i strani ključ (referencijalni integritet)

[11] engl. key constraints
[12] engl. foreign key constraints
[13] engl. referential integrity constraints

Prema *Definiciji 2.3*, vidjeli smo da je ključ relacije atribut ili skup atributa koji jednoznačno određuje svaku n-torku u relaciji (tablici). Između svih mogućih izbora za ključ pojedine relacije, dizajner baze bira tzv. *primarni ključ*, pomoću kojeg se n-torka može pronaći u bazi tako da navedemo vrijednosti atributa koji čine primarni ključ. Za pronalaženje n-torki možemo koristiti bilo koji ključ relacije, no SUBP očekuje navođenje primarnog ključa. Neki od SUBP kreiraju indekse[14] na primarnom ključu, kako bi pretraživanje bilo efikasnije. Ključne riječi pomoću kojih neke atribute proglašavamo ključem su `PRIMARY KEY` (za primarni ključ) i `UNIQUE` (za jedinstvenu vrijednost). No, treba naglasiti da svaka relacija može imati jedan primarni ključ, te više jedinstvenih vrijednosti ili jedinstvenih kombinacija pojedinih atributa. U *Primjeru 3.1* vidimo atribut `drzavaID` kao primarni ključ relacije `drzava`.

Osim ključa koji osigurava jedinstvenost n-torki, potrebno je objasniti i pojam ključa koji se poziva na već postojeći primarni ključ neke relacije, odnosno ključa koji se poziva na već postojeće n-torke. Takav se ključ zove *strani ključ*, a može se definirati na dva načina:

1. Navođenjem riječi `REFERENCES` nakon imena i tipa atributa kod definiranja tablice, a potom ime tablice i naziv atributa (koji mora biti primarni ključ ili jedinstvena vrijednost u tablici koja ga sadrži) na koji prethodno navedeni atribut referencira.

2. Pomoću ključnih riječi

   `FOREIGN KEY (<atributi>) REFERENCES <tablica>(<atributi>).`

   Može se koristiti i za više atributa koji bi zajedno činili strani ključ.

Ovdje još dobro dođe napomenuti kako pojava `NULL` vrijednosti u atributu stranog ključa ne krši ograničenje stranog ključa. Međutim, pojava `NULL` vrijednosti za atribut primarnog ključa nije dozvoljena (jer primarni ključ određuje n-torku na jedinstven način).

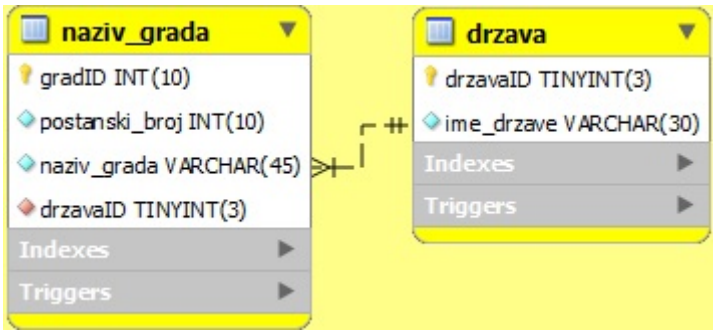
Primjer drugog načina definiranja stranog ključa pojavljuje se u radu u tablici `naziv_grada`, kada se atribut `drzavaID` iz tablice `naziv_grada` poziva na atribut `drzavaID` iz tablice `drzava`.

Primjer 3.2 (Definiranje stranog ključa) *Promotrimo relaciju* `naziv_grada` *(njezina je relacijska shema prikazana na* Slici 3.2*). Želimo dopustiti unošenje samo onih vrijednosti za atribut* `drzavaID` *koje već postoje za istoimeni atribut u tablici* `drzava`*. Kako bismo zadovoljili uvjet referencijalnog integriteta, u kreiranju tablice stavljamo ključne riječi*

`FOREIGN KEY (drzavaID) REFERENCES drzava (drzavaID).`

Pogledajmo sada definiciju tablice:

[14] O indeksima će više riječi biti kasnije, u *potpoglavlju 3.2.4*.



Slika 3.2 *Povezanost tablica* `naziv_grada` *i* `drzava`

```
CREATE TABLE IF NOT EXISTS naziv_grada (
  gradID INT UNSIGNED NOT NULL,
  postanski_broj INT UNSIGNED NOT NULL,
  naziv_grada VARCHAR(45) NOT NULL,
  drzavaID TINYINT UNSIGNED NOT NULL,
  PRIMARY KEY (gradID),
  CONSTRAINT drzavaID02
    FOREIGN KEY (drzavaID)
    REFERENCES drzava (drzavaID)
    ON DELETE RESTRICT
    ON UPDATE CASCADE)
ENGINE = InnoDB;
```

Ostalo nam je objasniti još nekoliko naredbi vezanih uz kod. Naredba `CONSTRAINT` *daje ime pojedinom ograničenju. U našem se slučaju ograničenje prethodno spomenutog stranog ključa zove* `drzavaID02`*. Ključne riječi* `ON DELETE` *nam govore što će se dogoditi s atributom* `drzavaID` *u tablici* `naziv_grada` *nakon brisanja retka iz tablice* `drzava`*, a ključna riječ on* `RESTRICT` *zabranjuje brisanje. Drugim riječima, nije dozvoljeno brisanje države u kojoj imamo naveden grad u bazi. Ključne riječi* `ON UPDATE` *nam govore što će se dogoditi s atributom* `drzavaID` *u tablici* `naziv_grada` *nakon ažuriranja atributa* `drzavaID` *u tablici* `drzava`*, a ključna riječ* `CASCADE` *znači kaskadiranje. Drugim riječima, kad bismo svim država u bazi povećali atribut* `drzavaID` *za* 500*, automatski bi se ažurirali istoimeni atributi u tablici* `naziv_grada` *(kaskadirano bi im se atribut* `drzavaID` *povećao za* 500*).*

Promotrimo sada još jednom dvije tablice koje smo prethodno naveli. Što bi moglo narušiti referencijalni integritet? Svaka od sljedeće četiri akcije:

- Unos retka tablice (n-torke) `naziv_grada` kojem atribut `drzavaID` nema `NULL` vrijednost, ali ne pripada nijednom atributu `drzavaID` iz tablice `drzava`.

- Promjena retka tablice (n-torke) `naziv_grada` nakon koje atribut `drzavaID` nema `NULL` vrijednost, ali ne pripada nijednom atributu `drzavaID` iz tablice `drzava`.
- Brisanje n-torke iz tablice `drzava` za koju postoji atribut `drzavaID` u tablici `naziv_grada` koji se referencira na njen `drzavaID`.
- Ažuriranje n-torke u tablici `drzava` koje mijenja atribut `drzavaID`, na koji se referencira atribut `drzavaID` iz tablice `naziv_grada`.

Dok prve dvije akcije zaista narušavaju referencijalni integritet, te ih kao takve sustav odbacuje, za zadnje dvije akcije imamo izbor (prema MySQL standardu):

- `RESTRICT` (zabrani) - odbija se bilo koja promjena koja krši referencijalni integritet; ne dopušta brisanje ili ažuriranje atributa `drzavaID` iz tablice `drzava` za koji postoji grad u tablici `naziv_grada` koji na njega referencira.
- `CASCADE` (kaskadiraj) - brisanjem države iz relacije `drzava` bi se automatski obrisali svi redci u relaciji `naziv_grada` kojima je atribut `drzavaID` jednak obrisanom atributu `drzavaID`, a promjenom (ažuriranjem) istog atributa, automatski bi se atributi `drzavaID` iz tablice `naziv_grada` ažurirali s njim.
- `SET NULL` - obrišemo li redak ili ažuriramo atribut `drzavaID` iz tablice `drzava` za koji postoji grad u tablici `naziv_grada` koji na njega referencira, automatski će se postaviti vrijednost atributa `drzavaID` iz tablice `naziv_grada` na `NULL`.

Vrijedi napomenuti da se svaka od te tri opcije u MySQL standardu može izabrati za brisanje (nakon riječi `ON DELETE` ide sintagma ključnih riječi pojedine akcije) i ažuriranje (sintagma ide nakon riječi `ON UPDATE`), kao što smo imali slučaj u *Primjeru 3.2.*

Izmjena ograničenja

Ograničenja možemo dodavati, mijenjati i brisati. Da bismo mijenjali ili obrisali pojedino ograničenje, nužno je dodijeliti mu ime. To se postiže pomoću ključnih riječi

`CONSTRAINT <ime ograničenja>`.

Imenovanje ograničenja pokazuje se dobrom praksom, jer omogućuje naknadne izmjene. Osim toga, u slučaju greške (prouzrokovane kršenjem ograničenja), dobit ćemo obavijest o kojem se ograničenju radi.

U *Primjeru 3.2* imamo ograničenje nazvano `drzavaID02` koje označava strani ključ. Podsjetimo se tog primjera:

```
CONSTRAINT drzavaID02
    FOREIGN KEY (drzavaID)
    REFERENCES drzava (drzavaID)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
```

Brisanje ograničenja se može postići pomoću ključnih riječi

`ALTER TABLE <ime tablice> DROP CONSTRAINT <naziv ograničenja>`.

Za naš prethodno naveden primjer, sintaksa je sljedeća:

```
ALTER TABLE naziv_grada DROP CONSTRAINT drzavaID02;
```

Želimo li ponovno uvesti prethodno obrisana ograničenja, ili dodati nova, koristimo sljedeće ključne riječi:

`ALTER TABLE <ime tablice> ADD CONSTRAINT <naziv ograničenja> <ograničenje>`.

Vrijedi napomenuti kako ograničenje mora biti udruženo s n-torkama, npr. ograničenje koje provjerava ključ ili ograničenje stranog ključa, te da ne možemo dodati ograničenje na tablicu ako ono ne vrijedi za svaku već postojeću n-torku u toj tablici.

Želimo li ponovno uvesti prethodno obrisano ograničenje, to postižemo na sljedeći način:

```
ALTER TABLE naziv_grada ADD CONSTRAINT drzavaID02
FOREIGN KEY (drzavaID)
    REFERENCES drzava (drzavaID)
    ON DELETE RESTRICT
    ON UPDATE CASCADE;
```

3.1.2 Okidači

Već smo u uvodu govora o ograničenjima i okidačima spomenuli kako su okidači poznati po tome što reagiraju na neki događaj. Zapravo, cijela priča o okidačima vezana je uz sintagmu *događaj-uvjet-akcija*[15]. Kad se događaj pojavi, provjeri uvjet; ako je istinit, poduzmi akciju. Zašto ih koristimo? Kako bismo premjestili logiku kontroliranja baze sa same aplikacije na SUBP[16], te kako bismo primijenili ograničenja (izvan onoga što sustav ograničenja podržava).

MySQL standard vezan uz okidače

[15] engl. event-condition-action (ECA)
[16] SUPB=sustav za upravljanje bazom podataka (engl. database management system)

U ovom radu ćemo govoriti o MySQL standardu[17] vezanom uz okidače. Najprije nekoliko riječi o kratici *događaj-uvjet-akcija*:

1. *Događaj* - to je ono što poziva ili pokreće pojedini okidač (najčešće je u pitanju unos, brisanje ili ažuriranje redaka pojedine relacije).

2. *Uvjet* - *boolean* izraz koji okidač provjerava nakon što se pojavio događaj. Ukoliko uvjet nije ispunjen, ništa se dalje ne događa iz okidača, a ako je ispunjen, izvršava se akcija.

3. *Akcija* - SQL iskaz(i). Akciju može činiti bilo koji niz operacija nad bazom, jedino u MySQL-u nije dopuštena akcija na tablici na kojoj se pojavio događaj koji pokreće okidač.

Sintaksa okidača u MySQL-u je sljedeća:

```
[delimiter <oznaka>]
CREATE TRIGGER ime_okidaca
BEFORE | AFTER dogadjaj
FOR EACH ROW
[BEGIN]
akcije
[END <oznaka>]
[delimiter ;]
```

Objasnimo sada svaki dio posebno:

- `CREATE TRIGGER ime_okidaca` - kreira okidač koji se naziva `ime_okidaca`
- `BEFORE | AFTER dogadjaj` - pokazuje događaj koji pokreće okidač i govori o tome na koji način okidač koristi stanje baze (tj. koristi li stanje u bazi prije nego što se događaj koji je pokrenuo okidač izvršio, ili stanje u bazi nakon samog događaja)
- `dogadjaj` - može biti unos retka ili redaka u neku relaciju, brisanje retka ili redaka iz neke relacije, ili njihovo ažuriranje. Ograničen je samo na cijelu n-torku, tj. `dogadjaj` = `INSERT | UPDATE | DELETE ON imetablice`.

[17] MySQL standard je implementacija SQL standarda vezanog uz okidače za MySQL SUBP. Više o samom SQL standardu može se naći na [3, str. 332-337]

- atribut starog[18] retka se označava sa `OLD.imeatributa`, a atribut novog[19] retka sa `NEW.imeatributa`, pri čemu `OLD.imeatributa` vrijedi samo za `DELETE` i `UPDATE` okidače, a `NEW.imeatributa` samo za `INSERT` i `UPDATE` okidače
- `FOR EACH ROW` - okidač na razini retka[20]. Na primjer, ako ažuriramo cijelu tablicu pomoću iskaza `UPDATE`, onda će se okidač na razini retka izvršiti jednom za svaki redak koji je ažuriran.
- `BEGIN...END` - u svakom okidaču može biti jedna ili više akcija. U slučaju više akcija, razdvojene su pomoću znaka ';', te okružene ključnim riječima `BEGIN...END` i u tom je slučaju potrebno postaviti neku drugu oznaku kao graničnik[21] (umjesto ';') - to postižemo naredbom `delimiter <oznaka>`, npr. `delimiter //` i nakon riječi `END` stavimo tu oznaku, a nakon okidača vratimo graničnik ';' (`delimiter ;`).
- `akcije` - zapravo je riječ o jednom ili više SQL iskaza

U radu smo koristili okidače najčešće za unos i ažuriranje podataka aktivne baze s povijesnom bazom. Okidač za unos bi imao naziv `imetablice_pov_ins`, a okidač za ažuriranje bi se zvao `imetablice_pov_upd`. Promotrimo takve primjere na relaciji `drzava`.

Primjer 3.3 (Okidač `drzava_pov_ins` u MySQL-u) *Navedimo najprije relacijsku shemu relacije* `drzava` *za potrebe ovog i sljedećeg primjera.*

`drzava(drzavaID, ime_drzave)`

Želimo svaku državu koja se unosi u aktivnu bazu ujedno upisati u povijesnu bazu. To postižemo sljedećim okidačem:

```
CREATE TRIGGER drzava_pov_ins
BEFORE INSERT ON aktivna.drzava
FOR EACH ROW
INSERT INTO povijesna.drzava VALUES(NEW.drzavaID, NEW.ime_drzave);
```

Objasnimo sada pojedine dijelove koda. Okidač `drzava_pov_ins` *se aktivira prije unosa u tablicu* `drzava` *u aktivnoj bazi* `(BEFORE INSERT ON aktivna.drzava)`*, te za svaki unešeni*

[18]Stari redak nam, kad se radi o operaciji brisanja, označava onaj redak koji događajem koji pokreće okidač brišemo. No, kada se radi o operaciji ažuriranja, stari redak označava ono stanje retka koje je bilo prije ažuriranja koje je pokrenulo okidač. Stari redak se koristi samo kod operacija brisanja i ažuriranja, nema ga kad se radi o unosu.

[19]Novi redak nam, kad se radi o operaciji unosa, označava onaj redak koji se unosi operacijom koja pokreće okidač. No, kada se radi o operaciji ažuriranja, novi redak označava stanje retka nakon ažuriranja koje je pokrenulo okidač

[20]engl. row-level trigger

[21]engl. delimiter

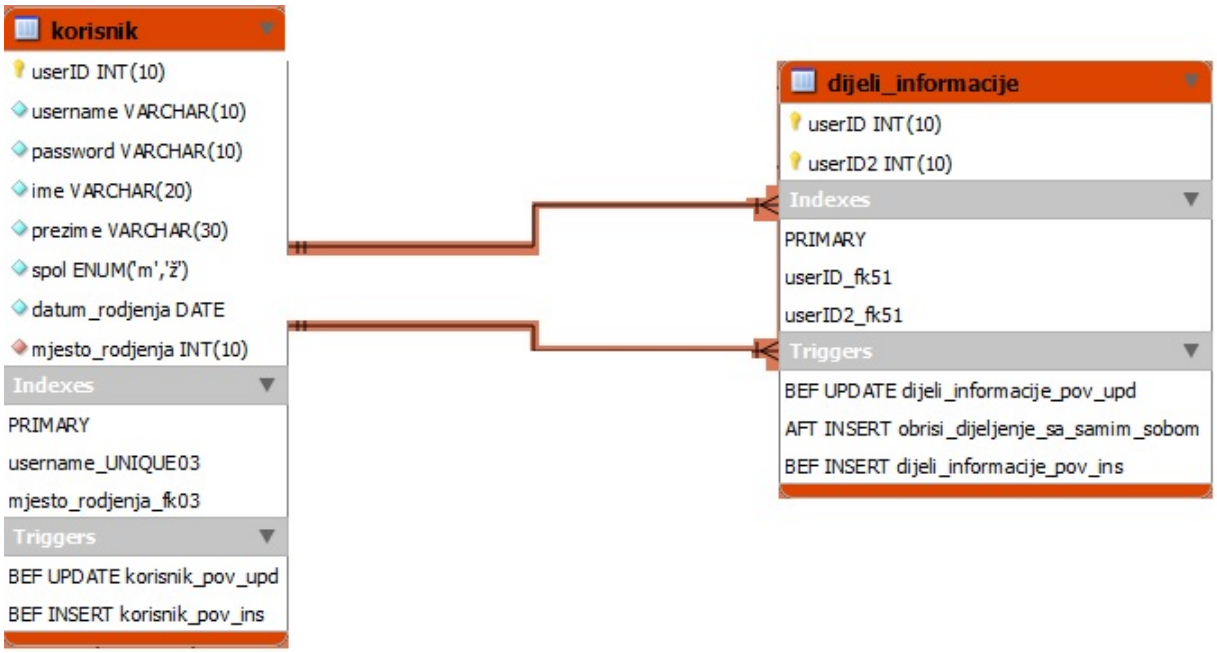
redak (FOR EACH ROW) *unosi vrijednosti retka u tablicu* drzava *u povijesnoj bazi. Ključna riječ* NEW *uz ime atributa označava vrijednost koju će taj atribut imati nakon operacije unosa u aktivnoj bazi. Drugim riječima, u tablicu* drzava *u povijesnoj bazi unosimo prije unosa u istoimenu tablicu aktivne baze one vrijednosti atributa svakog unešenog retka koje bi ti atributi imali nakon unosa u tablicu* drzava *u aktivnoj bazi (i u tom primjeru vidimo kako se okidač izvršava prije izvršavanja radnje događaja zbog kojeg se pokrenuo).*

Primjer 3.4 (Okidač drzava_pov_upd **u MySQL-u)** *Želimo osigurati usklađenost aktivne i povijesne baze po pitanju ažuriranja tablica. Sljedeći okidač osigurava usklađenost tablice* drzava *u aktivnoj i u povijesnoj bazi:*

```
CREATE TRIGGER drzava_pov_upd
BEFORE UPDATE ON aktivna.drzava
FOR EACH ROW
UPDATE povijesna.drzava
SET drzavaID=NEW.drzavaID, ime_drzave=NEW.ime_drzave
WHERE drzavaID=OLD.drzavaID;
```

Objasnimo sada pojedine dijelove koda. Okidač drzava_pov_upd *se aktivira prije ažuriranja redaka u tablici* drzava *u aktivnoj bazi* (BEFORE UPDATE ON aktivna.drzava)*, te za svaki ažurirani redak* (FOR EACH ROW) *mijenja vrijednosti istog retka u tablici* drzava *u povijesnoj bazi. Ključna riječ* NEW *uz ime atributa označava vrijednost koju će taj atribut imati nakon operacije ažuriranja u aktivnoj bazi, dok ključna riječ* OLD *uz ime atributa označava vrijednost koju taj atribut ima prije iste operacije ažuriranja. Drugim riječima, akcija okidača je ažuriranje retka tablice* drzava *u povijesnoj bazi onim vrijednostima atributa koje bi trebao poprimiti za atribute koji se ažuriraju u aktivnoj bazi istoimene tablice (a okidač se izvršava prije tog događaja). Okidač prepozna redak koji treba ažurirati po ključu* (WHERE drzavaID=OLD.drzavaID;)*.*

Primjer 3.5 (Okidači koji brišu dijeljenje informacija sa samim sobom) *Budući da MySQL ne podržava brisanje podataka iz tablice na kojoj je događaj pokrenuo okidač, morali smo napraviti dva okidača, jedan na unos retka u tablicu aktivne baze koja sadrži zapise o dijeljenju informacija* (AFTER INSERT ON aktivna.dijeli_informacije) *briše retke iz povijesne baze u kojima korisnici dijele informacije sami sa sobom* (DELETE FROM povijesna.dijeli_informacije WHERE userID=userID2;)*. A nakon brisanja iz tablice povijesne baze koja sadrži podatke o dijeljenju informacija* (AFTER DELETE ON povijesna.dijeli_informacije)*, aktivira se okidač koji iz aktivne baze briše podatke o osobama koje su same sa sobom podijelile informacije* (DELETE FROM



Slika 3.3 *Relacijske sheme relacija* `korisnik` *i* `dijeli_informacije`*, te veze između te dvije relacije*

`aktivna.dijeli_informacije WHERE userID=userID2;`). *Ovdje ujedno vidimo i primjer ulančanih okidača.*

```
USE aktivna;
CREATE TRIGGER obrisi_dijeljenje_sa_samim_sobom
AFTER INSERT ON aktivna.dijeli_informacije
FOR EACH ROW
DELETE FROM povijesna.dijeli_informacije
WHERE userID=userID2;

USE povijesna;
CREATE TRIGGER obrisi_dijeljenje_sa_samim_sobom
AFTER DELETE ON povijesna.dijeli_informacije
FOR EACH ROW
DELETE FROM aktivna.dijeli_informacije
WHERE userID=userID2;

USE aktivna;
```

Svaki okidač se može obrisati naredbom `DROP TRIGGER [imebaze.]<ime okidača>`. U

našim primjerima, to bi bilo ovako:

```
DROP TRIGGER drzava_pov_ins;
DROP TRIGGER drzava_pov_upd;
DROP TRIGGER obrisi_dijeljenje_sa_samim_sobom;
DROP TRIGGER povijesna.obrisi_dijeljenje_sa_samim_sobom;
```

Korisne napomene o okidačima u MySQL-u

Okidači u MySQL-u imaju nekoliko ograničenja. Ovdje ćemo navesti samo ona vezana uz rad, a više o ograničenjima u MySQL-u se može pronaći na [11] i [12]:

- U MySQL-u je dopušten[22] samo jedan okidač vezan uz pojedino vrijeme i pojedinu akciju za određenu tablicu. Npr. ne možemo imati dva okidača koja će se aktivirati nakon unosa retka u neku tablicu.
- MySQL ne podržava izvršavanje akcija okidača na tablici na kojoj je događaj pokrenuo okidač.
- Greška tijekom `BEFORE` ili `AFTER` okidača rezultira neuspjehom cijelog događaja koji je pokrenuo okidače.

Potrebno je napomenuti koliko je važno biti oprezan s korištenjem okidača, jer ulančane aktivnosti okidača i nepredvidljivi redoslijed u kojem SUBP izvršava aktivne okidače, može otežati razumijevanje efekta korištenja više okidača.

3.2 Prikazi i indeksi

Do sada smo govorili samo o tablicama - *temeljnim relacijama*[23]. No vrijedi spomenuti kako osim tog pojma, pod pojmom relacija možemo misliti i na druga dva pojma: na virtualne relacije (prikaze)[24] i na privremene relacije[25](privremene tablice). U ovom poglavlju, upoznat ćemo se s prikazima. Prikazi su relacije kojima su shema i sadržaj opisani kao SQL upit nad temeljnim relacijama ili nad već postojećim prikazima. Nisu pohranjeni u bazi, ali se

[22]Ovo ograničenje je vrijedilo do verzije 5.7.2. U njoj se može dogoditi da se na isti događaj u istoj tablici aktivira više okidača, ali se mora naznačiti redoslijed. To se postiže tako da se u okidaču nakon ključnih riječi `FOR EACH ROW` napiše `[PRECEDES | FOLLOWS] imeDrugogOkidaca`, čime se određuje hoće li se ovaj okidač dogoditi prije ili poslije okidača koji se zove `imeDrugogOkidaca`.

[23]engl. base relation - trajno pohranjena relacija (tablica)

[24]engl. virtual relations (views)

[25]engl. temporary relation (temporary table) - relacija koja nije prikaz, a nije ni trajno spremljena, nego može biti konstrurana za neki podupit i sl.

mogu izvršiti upiti nad njima, kao da postoje, jer će procesor upita[26] zamijeniti prikaz njegovom definicijom tako da izvrši upit iz prikaza. Sadržaj virtualnog prikaza se dinamički[27] određuje u samom trenutku kad se nad virtualnom relacijom izvrši neka operacija (jer ovisi o trenutnom stanju temeljnih tablica).

Možemo promatrati 3 sloja baze podataka - *fizički*, *konceptualni* i *logički*. Podaci pohranjeni na disk čine fizički sloj, organizacija podataka na disku u relacije čini konceptualni sloj, a prikazi čine logički sloj.

Zašto koristimo prikaze? Navedimo dva najčešća razloga. Jedan je mogućnost sakrivanja pojedinih podataka od pojedinih korisnika. Drugi razlog je učiniti neke upite jednostavnijima i prirodnijima.

Pored prikaza, u ovom ćemo poglavlju govoriti i o indeksima - strukturi podataka koja služi kako bi se pristup određenim redcima ubrzao. Bit će riječi i o tome koje sve činjenice treba uzeti u obzir vezano uz izbor indeksa za pojedinu tablicu.

3.2.1 Virtualni prikazi

Sintaksa za prikaz u MySQL-u je:

```
CREATE [OR REPLACE] VIEW ime_prikaza [(lista_atributa)]
AS definicija_prikaza
[WITH [CASCADED | LOCAL] CHECK OPTION];
```

Objasnimo sada pojedine dijelove (ovaj je dio uglavnom preuzet iz [12]):

- `CREATE VIEW` - stvara novi prikaz
- `CREATE OR REPLACE VIEW` - stvara novi prikaz, ako prikaz ne postoji (djeluje kao `CREATE VIEW`), ili mijenja postojeći (djeluje kao `ALTER VIEW`)
- `ime_prikaza` - prikaz ne smije imati isto ime kao bilo koja tablica ili prikaz u bazi
- `definicija_prikaza` - počinje sa `SELECT` klauzulom i moraju vrijediti sljedeća ograničenja:
  - `SELECT` iskaz ne smije sadržavati upit u `FROM` klauzuli, a ne smije se odnositi na korisničke varijable.

[26] engl. query processor

[27] tj. SUBP dinamički određuje sadržaj virtualnih relacija pri izvršavanju svakog upita koji koristi tu virtualnu relaciju

- Prikaz je "zamrznut" pri stvaranju, pa promjene na tablici koja je u pozadini ne utječu na definiciju prikaza (npr. prikaz koji u definiciji ima `SELECT *` za tablicu neće prikazivati stupce koji su naknadno dodani u tu tablicu).
- Svaka tablica ili prikaz koji su u definiciji prikaza moraju postojati u trenutku kad se prikaz kreira; međutim, nakon stvaranja prikaza moguće je obrisati tablicu ili prikaz na koje definicija upućuje. U tom slučaju, korištenje prikaza dovodi do greške. Kako bismo provjerili ima li definicija prikaza problem takve vrste, možemo koristiti SQL iskaz `CHECK TABLE <ime tablice ili prikaza>`, koji će javiti postoji li određena tablica ili prikaz na koje se pozivamo u definiciji prikaza, ili ne postoji.
- Definicija prikaza se ne može pozivati na privremenu tablicu (`TEMPORARY TABLE`) i ne može stvoriti privremeni prikaz (`TEMPORARY VIEW`).
- Prikaz može biti napravljen pomoću više `SELECT` klauzula. Može upućivati na temeljne tablice ili ostale prikaze, može koristiti spajanje, uniju ili podupite. `SELECT` se čak ne mora pozivati ni na jednu tablicu (npr. može računati neki izraz i tako nazvati stupac), a može se raditi i o izrazima koji koriste funkcije, konstantne vrijednosti, operatore i sl.
- Alias za ime stupca u `SELECT` iskazu se provjerava na maksimalnu duljinu stupca (64 znaka), a ne na maksimalnu duljinu aliasa (256 znakova).

- `ORDER BY` je dopušteno u definiciji prikaza, ali se ignorira ako selekcija u prikazu ima vlastiti `ORDER BY`
- `WITH CHECK OPTION` se koristi za izmjenjive[28] prikaze, kako se ne bi dopustio unos ili ažuriranje redaka, osim onih za koje je `WHERE` klauzula u definiciji prikaza istinita. Pri tome imamo izbor:
  - `LOCAL` - ograničava `CHECK OPTION` samo na onaj prikaz koji se definira,
  - `CASCADED` - provjerava evaluaciju i ishodišnih prikaza, a ne samo onog koji trenutno stvaramo; podrazumijeva se kao zadana opcija ako smo stavili ključne riječi `WITH CHECK OPTION`, a nismo naveli nijednu od riječi `LOCAL` ili `CASCADED`.

U samom radu, prikaze smo koristili najviše u slučajevima kada je trebalo povezati tablice, kako bismo mogli prikazati informacije. Ukoliko prikaz postoji za tablicu `imetablice`, on se u radu naziva `imetablice_prikaz`. Ovdje ćemo kao primjer navesti prikaz `naziv_grada_prikaz`, koji umjesto atributa `drzavaID` iz relacije `naziv_grada` ima atribut

[28] O izmjenjivim prikazima u MySQL-u će biti više riječi u *potpoglavlju 3.2.2*, a primjer spomenute opcije nalazi se u *potpoglavlju 3.2.3*.

ime_drzave iz relacije drzava. No, prije samog primjera, moramo navesti definiciju prirodnog spoja kako bismo mogli objasniti primjer.

Definicija 3.1 (Prirodni spoj dvaju tablica s istoimenim atributima) *Neka je* R(A,B) *relacija u kojoj je skup atributa* A *međusobno disjunktan skupu atributa* B*, te* S(B,C) *relacija u kojoj je skup atributa* B *međusobno disjunktan skupu atributa* C*, pri čemu su skupovi atributa* A *i* C *međusobno disjunktni.* ***Prirodni spoj*** RS *relacija* R *i* S *će imati relacijsku shemu* RS(A,B,C) *i sadržavat će sve retke nastale spajanjem redaka relacije* R *i redaka relacije* S *koji se podudaraju u vrijednostima zajedničkog skupa atributa* B*.*

Primjer 3.6 (Prikaz države uz svaki navedeni grad - naziv_grada_prikaz) *Želimo uz svaki grad upisan u bazu imati navedeno ime države umjesto atributa* drzavaID*. Za potrebe ovog primjera, ponovimo relacijske sheme temeljnih tablica koje se nalaze u definiciji prikaza:*

drzava(drzavaID, ime_drzave)
naziv_grada(gradID, postanski_broj, naziv_grada, drzavaID)

To ćemo postići tako da napravimo prirodni spoj tablice drzava *i tablice* naziv_grada*, a potom izaberemo sve atribute, osim atributa* drzavaID*:*

```
CREATE VIEW naziv_grada_prikaz AS
SELECT  gradID, postanski_broj, naziv_grada, ime_drzave
FROM drzava NATURAL JOIN naziv_grada;
```

Upiti se mogu izvršiti nad prikazima, kao i nad pohranjenim tablicama. Ime prikaza spomenemo u FROM klauzuli, oslanjajući se na SUBP, koji će doći do određenih n-torki operacijama nad relacijama iz definicije virtualnog prikaza.

Primjer 3.7 (Prikaz korisnik_prikaz koji navodi mjesto rođenja korisnika) *Želimo uz svakog korisnika imati naveden poštanski broj i naziv grada u kojem je rođen, te ime države. Za potrebe ovog prikaza ponovimo relacijsku shemu relacije* korisnik *sa* Slike 3.3*:*

korisnik(userID, username, password, ime, prezime, spol, datum_rodjenja, mjesto_rodjenja)

te uz to navedemo relacijsku shemu prethodno stvorenog ***prikaza*** naziv_grada_prikaz*:*

naziv_grada_prikaz(gradID, postanski_broj, naziv_grada, ime_drzave).

Prikaz koji ovdje navodimo spaja retke tablice korisnik *i prikaza* naziv_grada_prikaz *na način da izjednačava atribute* mjesto_rodjenja *iz relacije* korisnik *i* gradID *iz relacije* naziv_grada_prikaz (korisnik JOIN naziv_grada_prikaz ON

`korisnik.mjesto_rodjenja=naziv_grada_prikaz.gradID`), *te iz njih navodi atribute koje tražimo. Ključna riječ* `AS` *služi za preimenovanje atributa, npr. atribut* `postanski_broj` *će se u prikazu zvati* `postanski_broj_rodnog_grada`. *Donosimo sada kod prikaza.*

```
CREATE VIEW korisnik_prikaz AS
SELECT userID, username, ime, prezime, spol, datum_rodjenja,
postanski_broj AS 'postanski_broj_rodnog_grada',
naziv_grada AS 'grad_rodjenja', ime_drzave AS 'drzava_rodjenja'
FROM korisnik JOIN naziv_grada_prikaz
ON korisnik.mjesto_rodjenja=naziv_grada_prikaz.gradID;
```

Prikaz `korisnik_prikaz` *ima relacijsku shemu*

`korisnik_prikaz(userID, username, ime, prezime, spol, datum_rodjenja, postanski_broj_rodnog_grada, grad_rodjenja, drzava_rodjenja)`.

Pokažimo sljedećim primjerom na koji se način upit koji uključuje virtualni prikaz interpretira.

Primjer 3.8 (Interpretacija korištenja virtualnog prikaza u upitu drugog prikaza) *U ovom primjeru pokazujemo na koji način SUBP interpretira naš prethodno navedeni prikaz, koji u svojoj definiciji sadrži prikaz. SUBP napravi Kartezijev produkt svih temeljnih tablica (spoj koji sadrži sve atribute svake od relacija koje sudjeluju u spoju na način da svaki redak jedne relacije uparuje sa svakim retkom druge i da spoj sadrži sve moguće kombinacije redaka relacija koje u njemu sudjeluju) koje se koriste za definiciju prikaza* (*u našem slučaju relacija* `korisnik`, `drzava` *i* `naziv_grada`) *i potom traži samo one retke u kojima se atribut* `drzavaID` *iz tablice* `drzava` *podudara s istoimenim atributom iz tablice* `naziv_grada` *i u kojima se uz to atribut* `mjesto_rodjenja` *iz tablice* `korisnik` *podudara s atributom* `gradID` *iz tablice* `naziv_grada`. *SUBP interpretira prethodno navedeni prikaz kao da smo izvršili sljedeći kod.*

```
CREATE VIEW korisnik_prikaz AS
SELECT userID, username, ime, prezime, spol, datum_rodjenja,
postanski_broj AS 'postanski_broj_rodnog_grada',
naziv_grada AS 'grad_rodjenja', ime_drzave AS 'drzava_rodjenja'
FROM korisnik, drzava, naziv_grada
WHERE drzava.drzavaID=naziv_grada.drzavaID
AND korisnik.mjesto_rodjenja=naziv_grada.gradID;
```

U praksi, korisnik ne mora brinuti o tome na koji se način prikazi interpretiraju u SUBP. Želimo li ponekad dati drugačija imena atributima prikaza, možemo to učiniti i na način[29] da ih navedemo u zagradi nakon imena prikaza u `CREATE VIEW` iskazu.

Brisanje[30] prikaza ostvaruje se naredbom `DROP VIEW <ime prikaza>`, odnosno u našem slučaju:

```
DROP VIEW naziv_grada_prikaz;
DROP VIEW korisnik_prikaz;
```

3.2.2 Izmjena prikaza i izmjenjivi prikazi

Možemo li mijenjati prikaze?

U nekim slučajevima moguće je izvesti operacije unosa, brisanja ili ažuriranja na prikaz. Prvo logično pitanje koje se nameće uz tu tvrdnju je kako je to uopće moguće kad prikazi ne postoje u bazi pohranjeni kao što su to temeljne tablice. Što bi zapravo značilo unijeti novu n-torku u prikaz? Gdje bi ta n-torka završila, i kako bi se SUBP dosjetio da ta n-torka treba biti u prikazu?

Iako u dosta slučajeva nemoguće, postoje i tzv. **izmjenjivi prikazi**[31], u kojima je moguće izmjenu nad njima zapravo usmjeriti na izmjenu nad temeljnim tablicama (navedenima u prikazu).

Prije govora o izmjenjivim prikazima, napomenimo ponovno kako SUBP ne može promijeniti sadržaj prikaza, tj. retke tablice koja se formira kao rezultat SQL upita koji je u prikazu. Umjesto toga, SUBP mora promijeniti sadržaj temeljnih tablica koje se navode u prikazu. Ako je virtualni prikaz definiran tako da SUBP može jednoznačno odrediti koje operacije mora obaviti na temeljnim tablicama (umjesto na prikazima), tada kažemo da je prikaz izmjenjiv.

Izmjenjivi prikazi u MySQL-u

[29] `CREATE VIEW imePrikaza(naziv1, naziv2,..., nazivN) AS <definicija prikaza>`

[30] Važno je napomenuti kako taj SQL iskaz briše definiciju prikaza (tako da ne možemo više raditi upite nad prikazom, ili ga izmjenjivati), dok na relacijama koje je prikaz koristio nema promjena. S druge strane, ako obrišemo temeljne tablice (naredbom `DROP TABLE <ime tablice>`), prikazi koji se pozivaju na te tablice se neće automatski obrisati, ali će postati neupotrebljivi (budući da se pozivaju na nepostojeće relacije).

[31] engl. updatable views

Slijedi nekoliko informacija o izmjenjivim prikazima u MySQL-u, preuzeto iz [11] i [12].

Izmjenjivi prikazi su oni koje možemo koristiti u izrazima kao što su `DELETE`, `UPDATE`, `INSERT` kako bi se preko njih ažurirala temeljna tablica. Kako bi prikaz bio izmjenjiv, mora postojati veza 1-1 između redaka prikaza i redaka ishodišne temeljne tablice. Postoje slučajevi kada prikaz nije izmjenjiv. Točnije, prikaz nije izmjenjiv ako sadrži bilo što od sljedećeg:

- agregacijske funkcije (`SUM()`, `MIN()`, `MAX()`, `COUNT()`, `AVG()` i sl.),
- `DISTINCT`,
- `GROUP BY`,
- `HAVING`,
- `UNION, UNION ALL`,
- podupite u `SELECT` listi,
- određene spojeve,
- neizmjenjiv prikaz u `FROM` klauzuli,
- podupit u `WHERE` klauzuli koji se poziva na tablicu iz `FROM` klauzule,
- samo doslovne vrijednosti[32] (tada ne postoji ishodišna tablica za ažuriranje).

Uz navedeno, dodatni uvjeti da se u izmjenjiv prikaz mogu unijeti informacije (`INSERT`) su:

- ne smije postojati dvostruko ime za stupac prikaza,
- prikaz mora sadržavati sve stupce iz temeljne tablice koji nemaju zadanu vrijednost `DEFAULT`,
- stupci prikaza ne smiju biti izvedeni[33] iz stupaca temeljnih tablica.

U prikaz koji ima stupce iz temeljnih tablica i one izvedene se ne mogu unositi redci, ali se mogu ažurirati ako mijenjamo samo one stupce koji nisu izvedeni.

[32] eng. literal values, npr. atribut `stupac2` u prikazu `CREATE VIEW prikaz AS SELECT stupac, 1 AS stupac2 FROM tablica;`

[33] pojam izvedenog stupca označava stupce koji su rezultat nekog izraza, npr. zbroj, razlika, umnožak, i sl.

Primjer 3.9 (Unos u prikaz ne mora završiti u prikazu) *Za potrebe ovog prikaza, navedimo relacijsku shemu*[34] *relacije koja se nalazi u definiciji prikaza*

`grupa(IDgrupe, zipID, naziv, opis)`

Navedimo sada primjer prikaza `zipIDje1` *koji se tiče naziva i opisa grupa kojima je znanstveno-istraživačko područje jednako 1, te primjer unosa u njega.*

```
CREATE VIEW zipIDje1 AS
SELECT naziv, opis FROM grupa
WHERE zipID=1;


INSERT INTO zipIDje1 VALUES ('nova grupa','moj zipID=1');
```

Što će se dogoditi? Prije iznošenja te činjenice, želimo napomenuti kako ovaj primjer nemamo u samom radu, nego ga koristimo samo zbog teoretskih razloga. Naime, kako relacija `grupa` *ima atribute* `IDgrupe` *i* `zipID` *postavljene na* `NOT NULL`*, unos neće biti dozvoljen, jer bi, umjesto u prikaz, podaci trebali biti unešeni u tablicu* `grupa` *i to na ovaj način:* `(NULL, NULL, 'nova grupa','moj zipID=1')`*. No, maknemo li riječi* `NOT NULL` *kod definicije atributa* `IDgrupe` *i* `zipID` *u tablici* `grupa`*, tada će unos proći, jer prikaz* `zipIDje1` *zadovoljava uvjete (budući da sadrži komponente samo jedne temeljne tablice). Unos u prikaz* `zipIDje1` *je izvršen kao da je sljedeći unos naveden u tablicu* `grupa`*:*

```
INSERT INTO grupa(naziv, opis)
VALUES ('nova grupa','moj zipID=1');
```

Redak unešen u tablicu `grupa` *ima vrijednost* `'nova grupa'` *za atribut* `naziv`*, te vrijednost* `'moj zipID=1'` *za atribut* `opis`*. Atribut* `IDgrupe` *će poprimiti kao vrijednost automatski odabran broj koji slijedi nakon prethodno unešenog retka* `(AUTO_INCREMENT)`*, a atribut* `zipID` *će poprimiti vrijednost* `NULL`*. Uočimo što se sada događa. Unešeni redak, budući da ima* `NULL` *vrijednost za atribut* `zipID`*, neće prijeći u prikaz (koji traži da* `zipID` *ima vrijednost* 1*), pa sam unos ne utječe na prikaz. Kako bismo to popravili, možemo dodati atribut* `zipID` *u* `SELECT` *klauzuli prikaza.*

Primjer 3.10 (Popravak prikaza `zipIDje1`) *Poučeni prethodno navedenim primjerom kako unos retka u prikaz ne mora završiti u prikazu, preradimo prikaz* `zipIDje1` *tako da u njega (osim atributa navedenih u prethodnom primjeru) bude dodan i atribut* `zipID`*. Sada će unos retka u prikaz zahtijevati i unos atributa* `zipID`*, koji mora biti* 1*, pa će završiti u prikazu. Donosimo kod prikaza i primjer unosa.*

[34] Relacijska shema će se nalaziti i na *Slici 3.5.*

```
CREATE OR REPLACE VIEW zipIDje1 AS
SELECT zipID, naziv, opis FROM grupa
WHERE zipID=1;

INSERT INTO zipIDje1 VALUES (1, 'nova grupa','moj zipID=1');
```

Unos ima isti efekt na tablicu `grupa`*, kao da smo izvršili sljedeći redak:*

```
INSERT INTO grupa(zipID, naziv, opis)
VALUES (1, 'nova grupa','moj zipID=1');
```

Izmjenjivi prikazi omogućuju i brisanje. Odnosno brisanje se *prosljeđuje* do temeljne tablice `R`. Kako bismo bili sigurni da će se obrisati jedino n-torke u prikazu, dodajemo uvjet u `WHERE` klauzuli prikaza uvjetu u `WHERE` klauzuli za brisanje. Pokažimo to na primjeru.

Primjer 3.11 (Brisanje redaka iz prikaza) *Želimo obrisati iz izmjenjivog prikaza* `zipIDje1` *sve grupe koje u opisu imaju riječ* `'nova'`*. To ćemo učiniti na sljedeći način:*

```
DELETE FROM zipIDje1
WHERE naziv LIKE '%nova%';
```

Brisanje je proslijeđeno *do ekvivalentog brisanja u temeljnoj tablici* `grupa`*; jedina razlika je ta da je uvjet koji definira prikaz* `zipIDje1` *dodan uvjetu klauzule* `WHERE`*, pa je zapravo izvršena sljedeća naredba:*

```
DELETE FROM grupa
WHERE naziv LIKE '%nova%' AND zipID=1;
```

Na sličan način se i ažuriranje izmjenjivog prikaza prosljeđuje do temeljne tablice. Tako ažuriranje prikaza ima efekt ažuriranja svih n-torki iz temeljne relacije koje su se pojavile u prikazu.

Primjer 3.12 (Ažuriranje redaka iz prikaza) *Ažuriranje prikaza* `zipIDje1` *koje svim grupama iz prikaza daje opis* `zipID=1 u ovoj grupi`*, prikazano sljedećom sintaksom*

```
UPDATE zipIDje1
SET opis='zipID=1 u ovoj grupi';
```

je ekvivalentno sljedećem ažuriranju temeljne tablice `grupa`

```
UPDATE grupa
SET opis='zipID=1 u ovoj grupi'
WHERE zipID=1;
```

Prema [12], MySQL u trenutku stvaranja prikaza postavlja zastavicu, nazvanu zastavica izmjenjivosti prikaza[35]. Zastavica je postavljena potvrdno (`YES`) ili niječno (`NO`), ovisno o

[35] engl. view updatability flag

tome jesu li dozvoljene operacije ažuriranja, brisanja i sl. Stupac `IS_UPDATABLE` u tablici `INFORMATION_SCHEMA.VIEWS` pokazuje status zastavice. To znači da poslužitelj uvijek zna je li prikaz izmjenjiv. Ako prikaz nije izmjenjiv, operacije unosa, ažuriranja i brisanja će biti odbijene. Podsjetimo se: prikaz može biti izmjenjiv, a da u njemu nije dopuštena operacija unosa.

Za izmjenjiv prikaz koji se sastoji od više tablica, `INSERT` operacija se može provesti ako upisuje redak samo u jednu tablicu, a `DELETE` operacija nije podržana.

Želimo li vidjeti sve informacije o prikazima iz pojedine baze, rabimo sljedeći SQL upit:

```
SELECT * FROM INFORMATION_SCHEMA.VIEWS
WHERE TABLE_SCHEMA='imebaze';
```

Primjer 3.13 (Tražimo sve prikaze u bazi `aktivna`) *Želeći naći sve prikaze u bazi* `aktivna`*, tražimo uz svaki navedeno ime baze* `(TABLE_SCHEMA)`*, ime prikaza* `(TABLE_NAME)`*, definiciju prikaza* `(VIEW_DEFINITION)`*, ima li* `CHECK` *opciju* `(CHECK_OPTION)` *i je li izmjenjiv* `(IS_UPDATABLE)`*. To postižemo sljedećim upitom.*

```
SELECT TABLE_SCHEMA AS ime_baze,
TABLE_NAME AS ime_prikaza,
VIEW_DEFINITION AS definicija_prikaza,
CHECK_OPTION AS ima_li_CHECK_opciju,
IS_UPDATABLE AS je_li_izmjenjiv
FROM INFORMATION_SCHEMA.VIEWS
WHERE TABLE_SCHEMA='aktivna';
```

Definiciju prikaza koji postoji u trenutnoj bazi u MySQL-u možemo saznati i preko naredbe `SHOW CREATE VIEW <ime prikaza>`.

3.2.3 `CHECK` opcija vezana uz prikaze

U *Primjeru 3.9* smo vidjeli kako unos vrijednosti u prikaz može rezultirati unosom retka u temeljnu tablicu, koji se neće pojaviti u prikazu. Kako bi to spriječio, MySQL dopušta provjeru hoće li unos podataka u izmjenjivi prikaz ili njihova promjena i dalje ostati u prikazu. Spomenuta provjera se postiže pomoću ključnih riječi `WITH CHECK OPTION` nakon upita u prikazu. Promotrimo primjer na kojem ćemo objasniti o čemu se radi.

Primjer 3.14 (Osiguran unos samo onih redaka koji će završiti u prikazu) *Promotrimo prikaz* `zipIDje1` *koji ima jednaku definiciju prikaza kao istoimeni prikaz u* Primjeru 3.9*. Jedino se od njega razlikuje u ključnim riječima* `WITH CHECK OPTION`*.*

```
CREATE OR REPLACE VIEW zipIDje1 AS
SELECT naziv, opis FROM grupa
WHERE zipID=1
WITH CHECK OPTION;
```

Pokušajmo sada u njega unijeti novi redak

```
INSERT INTO zipIDje1 VALUES ('nova grupa3','moj zipID=1');
```

Nakon pokušaja unosa javit će se greška: `CHECK OPTION failed`*, jer navedeni redak ne bi završio u prikazu, budući da ima vrijednost* `NULL` *za atribut* `zipID`*, tako da unos retka u prikaz na ovaj način neće proći, za razliku od unosa u* Primjeru 3.9.

3.2.4 Indeksi u SQL-u

Sljedeća dva odlomka su prevedena iz [3, str. 350]:

"*Indeks* na atributu `A` relacije je struktura podataka koja čini uspješnim pronalaženje onih n-torki koje imaju fiksnu vrijednost atributa `A`. Indeks možemo promatrati kao pretraživanje binarnog stabla s parovima (ključ, vrijednost), u kojima je ključ a (jedna od vrijednosti koju atribut `A` mora imati) povezan s "vrijednošću" koja je skup lokacija n-torki koje imaju a u komponenti atributa `A`. Takav indeks može pomoći s upitima u kojima se atribut `A` uspoređuje s konstantom, npr. $\texttt{A} = 3$, ili čak $\texttt{A} \leq 3$. Uočimo kako ključ za indeks može biti bilo koji atribut ili skup atributa, tj. ne mora nužno biti ključ relacije na kojoj se indeks postavlja. Kada moramo razlikovati ta dva pojma, atribute indeksa možemo zvati *ključ indeksa*.

Tehnologija implementiranja indeksa na velikim relacijama je od temeljne važnosti u implementaciji SUBP. Najvažnija struktura podataka koju SUBP koristi po tom pitanju je *B-stablo*, koje je generalizacija balansiranog[36] binarnog stabla." U ovom dijelu rada nećemo se baviti detaljnije tematikom B-stabla (više o njima ima u [3, str. 633]).

Zašto govorimo o indeksima?

U slučajevima kada imamo velike relacije, može nam biti *skupo* proći svaku n-torku u relaciji kako bismo našli one (možda vrlo malo njih) koje zadovoljavaju dani uvjet.

Primjer 3.15 (Pronalaženje n-torke u relaciji) *Za potrebe ovog primjera, podsjetimo se najprije relacijske sheme relacije* `korisnik` *sa* Slike 3.3*:*

```
korisnik(userID, username, password, ime, prezime, spol, datum_rodjenja,
```

[36] Stablo je balansirano ako je duljina puta od korijena do lista jednaka za svaki list u stablu.

```
mjesto_rodjenja)
```

Pretpostavimo da u društvenoj mreži želimo naći one korisnike kojima ime počinje na slovo S, i koji su rođeni 1989. *godine:*

```
SELECT * FROM korisnik
WHERE ime LIKE 'S%' AND datum_rodjenja LIKE '1989%';
```

Možemo imati na tisuće korisnika od kojih bi možda stotinjak bilo rođeno 1989. *godine.*

Naivan način implementacije ovog upita je uzeti sve n*-torke korisnika (sve tisuće koje imamo) i testirati uvjet iz* `WHERE` *klauzule na svakoj. No, bilo bi efikasnije ako bismo na neki način mogli dohvatiti njih stotinjak koji su rođeni* 1989. *godine i testirati svaki od njih kako bismo vidjeli kojima ime počinje slovom S. Postoji i još efikasniji način od toga, na koji bismo mogli izravno dohvatiti onih* 10*-ak* n*-torki koje zadovoljavaju oba uvjeta (tj. kojima ime počinje na slovo S i koji su rođeni* 1989. *godine).*

Indeksi također mogu biti korisni kad koriste spajanje tablica. Sljedeći primjer pokazuje takav slučaj:

Primjer 3.16 (Primjer spajanja tablica u kojima bi indeks bio od koristi) *Relacijsku shemu relacije* `korisnik` *imali smo u prethodnom primjeru. Podsjetimo se uz to, za potrebe ovog primjera, relacijske sheme relacije* `naziv_grada`*:*

```
naziv_grada(gradID, postanski_broj, naziv_grada, drzavaID)
```

Pogledajmo sada primjer upita za koji bi indeks bio od koristi.

```
SELECT userID, username, ime, prezime, spol, datum_rodjenja,
postanski_broj AS 'postanski_broj_rodnog_grada',
naziv_grada AS 'grad_rodjenja',
FROM korisnik JOIN naziv_grada
ON korisnik.mjesto_rodjenja=naziv_grada.gradID
WHERE naziv_grada=Osijek;
```

Kada bi postojao indeks na atributu `naziv_grada` *relacije* `naziv_grada`*, mogli bismo koristiti taj indeks kako bismo dohvatili* n*-torku za* Osijek*. Od te* n*-torke možemo izdvojiti* `gradID`*. Sada, pretpostavimo kako također postoji indeks na* `mjesto_rodjenja` *relacije* `korisnik`*. Tada možemo koristiti* `gradID` *s indeksom kako bismo našli* n*-torku iz relacije* `korisnik` *koja odgovara korisnicima rođenim u Osijeku. Iz te* n*-torke izdvojimo informacije koje nas zanimaju. Uočimo kako s ta dva indeksa gledamo samo one dvije vrste redaka (po jednu iz svake relacije) koji odgovaraju na upit. Bez indeksa, morali bismo gledati svaki redak svake relacije.*

Stvaranjem indeksa na atributu `imeAtributa` relacije `imeRelacije` će procesor upita izvršavati upite u kojima je atribut `imeAtributa` naveden na način da će gledati samo one n-torke iz relacije `imeRelacije` s navedenom vrijednošću atributa `imeAtributa`. Kao posljedica toga, vrijeme potrebno za odgovor na upit se smanjuje.

Sintaksa za stvaranje indeksa u MySQL-u

Sintaksa za stvaranje indeksa na `InnoDB` tablicama[37] u MySQL-u je sljedeća:

```
CREATE [UNIQUE | FULLTEXT] INDEX imeIndeksa
ON imeTablice(lista_atributa);
```

Objasnimo sada pojedine dijelove (prema [12]):

- `CREATE INDEX` omogućuje dodavanje indeksa na postojeće tablice.
- Lista atributa `(atr1, atr2)` stvara indeks na više atributa. Ključ indeksa se formira konkatenacijom vrijednosti danih atributa.
- Indeks se može stvoriti tako da koristi samo početni dio vrijednosti stupca pomoću sintakse `imeStupca(duljina)` koja određuje duljinu prefiksa indeksa:
  - Prefiks se može odrediti za `CHAR`, `VARCHAR`, `BINARY` i `VARBINARY` stupce.
  - `BLOB` i `TEXT` stupce također možemo indeksirati, ali mora biti zadana duljina prefiksa.
  - Duljina prefiksa se daje u znakovima za nebinarne znakovne tipove, a u bajtovima za binarne znakovne tipove. Tj. unosi za indeks se sastoje od prvih `duljina` znakova za stupce s tipom `CHAR`, `VARCHAR` i `TEXT`, te za prvih `duljina` bajta za svaki stupac tipa `BINARY`, `VARBINARY`, i `BLOB`.
- Ako se imena stupaca obično razlikuju u prvih 10 znakova, indeks ne bi trebao biti puno sporiji od indeksa stvorenog na cijelom stupcu. Uz to, korištenje prefiksa za indekse nad stupcima može smanjiti indeks datoteku[38], što bi uštedjelo prostor na disku i također ubrzalo `INSERT` operacije.
- `UNIQUE` indeks stvara ograničenje takvo da sve vrijednosti indeksa moraju biti različite. Greška se javlja ukoliko pokušamo dodati novi redak s vrijednosti ključa koja već postoji ili ako pokušamo stvoriti indeks nad relacijom u kojoj ime atributa nema

[37] `InnoDB` tablice osiguravaju zadovoljenje referencijalnog integriteta, a od MySQL 5.5 verzije su zadani stroj za pohranu (engl. storage engine). Više o njima u [12, str. 1574]

[38] engl. index file

jedinstvenu vrijednost. `UNIQUE` indeks dozvoljava višestruke `NULL` vrijednosti za stupce koji mogu sadržavati `NULL` vrijednosti. Ako odredimo vrijednost prefiksa za stupac s `UNIQUE` indeksom, vrijednosti stupca moraju biti jedinstvene u tom prefiksu.

- `FULLTEXT` indeksi su podržani za `InnoDB` tablice i mogu sadržavati samo `CHAR`, `VARCHAR` i `TEXT` stupce. Indeksira se uvijek cijeli stupac, indeksiranje prefiksa stupca nije podržano (duljina prefiksa se ignorira ako je navedena). Dobro dođe za pretraživanje cijelog teksta pomoću nekoliko funkcija u MySQL-u. Više detalja može se pronaći u [12, str. 1606].
- Za sada se vrijednosti indeksa u MySQL-u se uvijek pohranjuju u rastućem redoslijedu.

Želimo li obrisati indeks u MySQL-u, koristimo sljedeću sintaksu:

```
DROP INDEX imeIndeksa ON imeTablice;
```

ili

```
ALTER TABLE imeTablice DROP INDEX imeIndeksa;
```

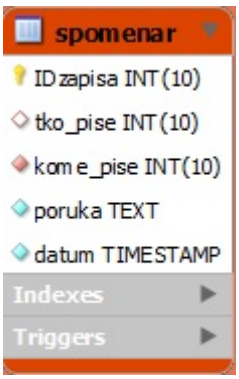
Primjer 3.17 (Stvaranje jedinstvenog indeksa) *U relaciji* `drzava` *s relacijskom shemom* `drzava(drzavaID, ime_drzave)` *želimo osigurati jedinstvenu vrijednost za* `ime_drzave` *i to postižemo na sljedeći način:*

```
CREATE UNIQUE INDEX ime_drzave_UNIQUE01 ON drzava(ime_drzave);
```

Primjer 3.18 (Stvaranje indeksa za pretraživanje teksta poruke) *U relaciji* `tekst_poruke` *s relacijskom shemom* `tekst_poruke(tekstID, tekst_poruke)` *želimo stvoriti indeks koji bi omogućio pretraživanje cijelog teksta poruke. To činimo na sljedeći način:*

```
CREATE FULLTEXT INDEX tekstPoruke16 ON tekst_poruke(tekst_poruke);
```

Kada ne bismo imali `FULLTEXT` *indeks, ne bismo mogli indeksirati cijeli tekst poruke, nego bismo morali navesti duljinu - prvih nekoliko znakova teksta poruke.*



Slika 3.4 *Relacijska shema relacije* `spomenar`

Primjer 3.19 (Indeksiranje s određenom duljinom znakova) *Promotrimo relacijsku shemu relacije* `spomenar` *sa* Slike 3.4. *Želimo osigurati jedinstvenost zapisa u spomenaru na način da će se zapis jedne osobe drugoj određenog datuma razlikovati u prvih 200 znakova poruke. To postižemo na sljedeći način:*

```
CREATE UNIQUE INDEX jedinstveni_zapis18 ON
spomenar(tko_pise, kome_pise, poruka(200), datum);
```

Primjer 3.20 (Stvaranje jednog indeksa na više atributa) *Budući da su* `fakultetID` *i* `profesorID` *ključ relacije* `web_stranice_prof`, *možemo očekivati da će se odrediti vrijednosti oba atributa, ili nijednog. Slijedi sintaksa kojom bismo deklarirali indeks na ova dva atributa u slučaju kad oni ne bi činili ključ, jer ovako je indeks već automatski stvoren u MySQL-u, ali ovdje nam je bitno zbog primjera imati ujedno i naziv indeksa:*

```
CREATE INDEX fakultetprofesor_fk31 ON
web_stranice_prof(fakultetID, profesorID);
```

Kako je `(fakultetID, profesorID)` *ključ, slijedi da za dane vrijednosti fakulteta i profesora indeks nalazi samo jedan redak - traženi redak. Usporedimo to sa slučajem kada upit određuje* `fakultetID` *i* `profesorID`, *ali imamo indeks samo na atributu* `fakultetID`. *Tada SUBP uzima u obzir sve profesore na navedenom fakultetu i provjerava tko je od njih onaj kome atribut* `profesorID` *odgovara vrijednosti navedenoj u upitu.*

Indeks `fakultetprofesor_fk31` *se koristi za upite koji su oblika:*

```
SELECT * FROM web_stranice_prof
WHERE fakultetID=broj1
AND profesorID=broj2;
```

i tome ekvivalentnog oblika:

```
SELECT * FROM web_stranice_prof
WHERE profesorID=broj2
AND fakultetID=broj1;
```

te za upite oblika:

```
SELECT * FROM web_stranice_prof
WHERE fakultetID=broj1;
```

za fiksno određene cjelobrojne vrijednosti `broj1` *i* `broj2`*, ali se ne može koristiti za:*

```
SELECT * FROM web_stranice_prof
WHERE profesorID=broj2;
```

Indeks `fakultetprofesor_fk31` *se efikasno koristi za sortiranja oblika:*

```
SELECT * FROM web_stranice_prof
ORDER BY fakultetID, profesorID;
```

```
SELECT * FROM web_stranice_prof
ORDER BY fakultetID DESC, profesorID DESC;
```

ali ne i za:

```
SELECT * FROM web_stranice_prof
ORDER BY fakultetID DESC, profesorID;
```

Osim toga, budući da imamo indeks na kombinaciji atributa `fakultetID` *i* `profesorID`*, nije nam potrebno stvoriti poseban indeks samo na atributu* `fakultetID`*.*

Ako je (što je čest slučaj u praksi) ključ indeksa koji sadrži više atributa zapravo konkatenacija atributa u određenom redoslijedu, onda čak možemo koristiti taj indeks kako bismo našli sve retke s danom vrijednošću za prvi atribut. Tako je dio dizajniranja indeksa koji se sastoji od više atributa izbor redoslijeda u kojem će se navesti atributi. Npr. ako želimo u upitu o web-stranicama profesora na fakultetima češće određivati fakultet, redoslijed atributa u indeksu će biti kao gore, a ako želimo češće određivati profesora čiju web-stranicu tražimo, redoslijed atributa u indeksu će biti `(profesorID, fakultetID)`*.*

Izbor indeksa

Iako možemo dobiti dojam kako što više indeksa stvorimo, veća je vjerojatnost da će za bilo koji upit postojati indeks koji nam je koristan, takav zaključak je kriv. Kako bismo to objasnili, spomenimo najprije činjenicu da su n-torke relacije pravilno pohranjene među stranicama[39] diska, a pregledavanje čak i jednog retka zahtjeva da cijela stranica bude u radnoj[40] memoriji. S druge strane, pregledavanje svih redaka na stranici malo više vremena košta od pregledavanja samo jednog retka. Cijena upita ili modifikacije je često broj stranica

[39] engl. pages of disk
[40] engl. main memory

koje trebamo dovesti u radnu memoriju, pa nam indeksi na određenom atributu mogu uvelike ubrzati izvođenje onih upita u kojima je vrijednost ili skup vrijednosti određen za taj atribut, te također mogu ubrzati spajanje tablica koje uključuje taj atribut. S druge strane, svaki indeks stvoren za jedan ili više atributa neke relacije čini kompleksnijim i vremenski složenijim operacije unosa, brisanja i ažuriranja na toj relaciji, zbog činjenice da i sami indeksi trebaju biti pohranjeni na disku. Zbog toga pristup indeksima i modifikacija indekasa košta. Zapravo, modifikacija je otprilike dvostruko skuplja od pristupa indeksu ili podacima u upitu, jer zahtjeva jedno čitanje stranice diska i osim toga jedno zapisivanje promijenjene stranice.

U praksi te činjenice određuju osigurava li dizajn baze zadovoljavajuće performanse.

Sljedeći primjer pokazuje moć indeksa za ključ, čak i u upitu koji uključuje spajanje.

Primjer 3.21 (Važnost indeksa kod uparivanja redaka) *Promotrimo sljedeći upit na relacije*
`drzava(drzavaID, ime_drzave)` *i*
`naziv_grada(gradID, postanski_broj, naziv_grada, drzavaID)`

```
SELECT *
FROM drzava, naziv_grada
WHERE drzava.drzavaID=naziv_grada.drzavaID
AND gradID=10;
```

U gore navedenom upitu događa se uparivanje redaka iz relacija `drzava` *i* `naziv_grada`*. Implementacija spoja na takav način zahtjeva od nas čitanje svake od stranica koja sadrži* n*-torke relacije* `drzava` *i svake od stranica koja sadrži* n*-torke relacije* `naziv_grada` *barem jednom. U stvari, budući da te stranice mogu biti previše brojne da bi stale u radnu memoriju u isto vrijeme, možda ćemo više puta morati čitati svaku stranicu s diska. No, imamo li odgovarajuće indekse, cijeli upit može biti gotov čitanjem* 2 *stranice.*

Indeks na ključu `gradID` *relacije* `naziv_grada` *će nam pomoći naći onaj redak kojem je* `gradID` *jednak* 10*. Samo ta stranica - stranica koja sadrži taj redak će biti pročitana s diska. Potom, nakon pronalaženja tog retka, dohvaćamo broj za njegov atribut* `drzavaID`*. Indeks na atributu* `drzavaID` *relacije* `drzava` *će nam pomoći brzo pronaći onaj redak koji odgovara državi u kojoj se nalazi pronađeni grad iz relacije* `naziv_grada`*. Ponovno, samo jedna stranica s retcima relacije* `drzava` *će biti pročitana s diska, iako možemo trebati pročitati mali broj drugih stranica za korištenje indeksa na atributu* `drzavaID` *relacije* `drzava`*.*

Još neke napomene vezane uz indekse

U ovom potpoglavlju, navest ćemo još neke korisne informacije o indeksima, na koje se nismo prije mogli toliko fokusirati. Rekli smo kako indeks ima strukturu B-stabla. To se B-stablo formira kao struktura nad podacima relacije izvršavanjem naredbe za stvaranje indeksa nad tom relacijom.

Imaju li vrijednosti atributa relativno mali broj različitih vrijednosti, indeks se ne bi trebao kreirati (npr. atribut `spol` u relaciji `korisnik`). Trebamo li u relaciji izvršiti mnogo operacija unosa, ažuriranja ili brisanja n-torki, preporuča se brisanje postojećih indeksa i njihovo ponovno kreiranje nakon tih operacija. Sadrži li relacija relativno malo redaka, nije potrebno stvarati indeks, jer B-stablo ne pridonosi efikasnosti pretrage.

Imamo li u nekoj relaciji atribut u kojem relativno malo redaka ima određenu vrijednost tog atributa, indeks bi mogao biti efikasan jer za pronalaženje tog atributa ne bismo trebali dopremiti sve stranice diska u radnu memoriju (nego samo one koje sadrže redak s tom vrijednošću atributa). Ukoliko su retci relacije sa zajedničkom vrijednošću nekog atributa grupirani na minimalno stranica diska koliko je moguće, indeks na tom atributu bi i u tom slučaju bio od koristi.

3.2.5 Kako MySQL koristi indekse

Za potrebe ovog cijelog potpoglavlja, koristili smo [12].

Većina MySQL indeksa su pohranjeni kao B-stabla (npr. `PRIMARY KEY, UNIQUE, FULLTEXT, INDEX`).

MySQL koristi indekse za sljedeće operacije:

- za brzi pronalazak redaka koji zadovoljavaju uvjet u `WHERE` klauzuli,
- za eliminiranje redaka iz izbora (ako imamo izbor između višestrukih indeksa, MySQL redovito koristi indeks koji pronalazi najmanji broj redaka),
- za vraćanje[41] redaka iz drugih tablica kada izvodi spojeve[42],
- za pronalazak najmanje (`MIN()`) i najveće (`MAX()`) vrijednosti određenog indeksiranog stupca,

[41] engl. retrieve

[42] MySQL može efikasnije koristiti indekse na stupcima ako su deklarirani kao isti tip i ista veličina - npr. `CHAR` i `VARCHAR` se u tom kontekstu smatraju jednakima ako su deklarirane kao jednake veličine (`VARCHAR(20)` i `CHAR(20)`).

- za sortiranje ili grupiranje tablice, ako je sortiranje ili grupiranje izvedeno po atributima ključa, od lijevo navedenog (npr. `ORDER BY atr1, atr2`).

Ukoliko nakon svih atributa ključa slijedi `DESC`, ključ se čita u obrnutom redoslijedu.

Indeksi su manje važni za upite na malim tablicama, ili na velikim tablicama gdje upit prolazi većinu redaka ili sve. Kada upit mora pristupiti većini redaka, skeniranje svih redaka je brže nego rad s indeksom - minimizira pretraživanje diska.

Primarni ključ ima automatski generiran indeks koji je s njim povezan. Ne moramo posebno dodati indeks na njega u MySQL-u.

Primarni indeks tablice trebao bi biti što je moguće kraći (kako bi se svaki redak mogao jednostavno i jasno identificirati).

3.3 Pohranjene procedure i funkcije

U ovom poglavlju rada bavit ćemo se tematikom pohranjenih procedura i pohranjenih funkcija. Oboje su potprogrami koji se izvršavaju u kontekstu SUBP, ali spomenimo razliku između ta dva pojma. Procedura u pozivajući program ne vraća rezultat, dok funkcija u pozivajući program vraća rezultat. U MySQL-u moguće je izvoditi naredbe za kontrolu toka programa (`IF`, `FOR`, `WHILE`,...), te koristiti varijable. Njima se može omogućiti zaštita podataka na razini retka[43] (radi se o mogućnosti da neki korisnik vidi samo neke od redaka u pojedinoj relaciji, a ne sve), a kod potreban za neki postupak, iskaz i sl. je napisan na jednom mjestu. U ovom radu opisat ćemo MySQL standard.

3.3.1 Kreiranje pohranjenih procedura i pohranjenih funkcija

Sintaksa procedure u MySQL-u je

```
CREATE PROCEDURE ime ([parametri_procedure[,...]])
tijelo_procedure;
```

a sintaksa funkcije

```
CREATE FUNCTION ime ([parametri_funkcije[,...]])
RETURNS tip
tijelo_funkcije;
```

[43] engl. row-level security

Parametri procedure se navode na sljedeći način `[IN | OUT | INOUT] <ime_parametra> <tip>`, a **parametri funkcije** `<ime_parametra> <tip>`.

Popis parametara se nalazi unutar zagrada. Ukoliko nemamo parametara, mora biti prisutna prazna zagrada `()`. Imena parametara nisu osjetljiva na malo/veliko[44] slovo.

Objasnimo sada što svaka od vrsta[45] parametara znači:

- `IN` (samo unos[46]) parametar prosljeđuje vrijednost proceduri, a u slučaju promijene vrijednosti parametra (unutar procedure), ne može ju vidjeti pozivatelj procedure.
- `OUT` (samo izlaz[47]) parametar prosljeđuje vrijednost iz procedure natrag pozivatelju, a početna vrijednost mu je `NULL`.
- Parametar `INOUT` (i ulaz i izlaz) je inicijaliziran od pozivatelja, može biti izmijenjen u proceduri, vidljiv je pozivatelju kad procedura završi.

Zadana vrsta parametra procedure je `IN` i može se izostaviti.

S druge strane, parametri funkcije mogu biti jedino `IN`, a jedini način na koji dobivamo informaciju iz funkcije je preko njezine vraćene vrijednosti. Ne smijemo određivati `IN` vrstu za parametre funkcije, iako to činimo u definicijama procedura.

Procedura se u MySQL-u briše naredbom
`DROP PROCEDURE <imeProcedure>;`

a funkcija naredbom
`DROP FUNCTION <imeFunkcije>;`

Navedimo sada primjer procedure, kako bismo mogli govoriti o načinu pozivanja procedure s obzirom na vrstu parametara koji su u njoj zadani.

Primjer 3.22 (Korištenje procedure za pretraživanje prikaza) *Promotrimo proceduru koja daje podatke o korisnicima iz* ***prikaza***

```
korisnik_prikaz(userID, username, ime, prezime, spol, datum_rodjenja,
postanski_broj_rodnog_grada, grad_rodjenja, drzava_rodjenja)
```

[44] engl. case sensitive
[45] engl. mode
[46] engl. input-only
[47] engl. output-only

za učitano ime i prezime korisnika.

```
CREATE PROCEDURE pretrazivanje_po_imenu_i_prezimenu(
IN imeKorisnika VARCHAR(20),
IN prezimeKorisnika VARCHAR(30))
SELECT * FROM korisnik_prikaz WHERE ime=imeKorisnika
AND prezime=prezimeKorisnika;
```

Uočimo kako su oba parametra `imeKorisnika` *i* `prezimeKorisnika` *vrste* `IN` *i tipa* `VARCHAR` `(VARCHAR(20)` *za* `imeKorisnika`*, jer je i atribut* `ime` *relacije* `korisnik_prikaz` *također tipa* `VARCHAR(20)`*, i* `VARCHAR(30)` *za* `prezimeKorisnika` *jer je atribut* `prezime` *relacije* `korisnik_prikaz` *tipa* `VARCHAR(30))`*. Uočimo kako se imena parametara u tijelu procedure mogu koristiti kao da su konstante. Uz to, na ovakav način korisnik neće vidjeti sve retke iz prikaza* `korisnik_prikaz`*, nego samo one kojima je ime jednako učitanom imenu, a prezime jednako učitanom prezimenu, pa možemo reći da je osigurana zaštita podataka na razini retka.*

Procedura se poziva sljedećim ključnim riječima:

```
CALL <ime procedure>(<popis argumenata>);
```

Procedura iz *Primjera 3.22* bi se pozvala na način:

```
CALL pretrazivanje_po_imenu_i_prezimenu('Snježana','Mijošević');
```

Napomenimo kako nije dozvoljeno pozivanje funkcije naredbom `CALL`. Funkcija se poziva pomoću

```
SELECT <ime funkcije>(<popis argumenata>);
```

Za svaki `OUT` ili `INOUT` parametar, korisnik prosljeđuje varijablu u `CALL` iskaz koji poziva proceduru pa završetkom procedure možemo dobiti njenu vrijednost. Promotrimo takav slučaj na sljedećem primjeru.

Primjer 3.23 (Primjer pozivanja procedure s `OUT` parametrom) *Sada na primjeru procedure koja nalazi broj ljudi koji sa zadanim korisnikom dijeli informacije i ispisuje iz prikaza korisnika koji su to ljudi pokažimo kako pozvati proceduru s* `OUT` *parametrom. Podsjetimo se najprije relacijskih shema* ***prikaza*** `korisnik_prikaz` *i* ***relacije*** `dijeli_informacije`

```
korisnik_prikaz(userID, username, ime, prezime, spol, datum_rodjenja,
postanski_broj_rodnog_grada, grad_rodjenja, drzava_rodjenja)

dijeli_informacije(userID, userID2)
```

Tijelo procedure ima 2 SQL iskaza pa su potrebne ključne riječi `BEGIN` *i* `END`*, te promjena graničnika prije i nakon procedure* (`delimiter // ... delimiter ;`)*.* `IN` *parametar* `mojID` *označava* `userID` *korisnika za kojeg tražimo koliko ljudi s njim dijeli informacije i koji su to ljudi, a u* `OUT` *parametar* `broj` *(za kojeg će u pozivanju procedure biti prosljeđena varijabla) će se pomoću klauzule* `INTO broj` *upisati broj korisnika koji je s našim korisnikom podijelio informacije. Ispis koji korisnici su s našim korisnikom podijelili informacije će biti poredan po korisničkom broju* (`ORDER BY userID`)*. Promotrimo sada kod naše procedure.*

```
delimiter //
CREATE PROCEDURE koliko_ljudi_sa_mnom_dijeli_informacije (
IN mojID INT UNSIGNED,
OUT broj TINYINT UNSIGNED)
BEGIN
SELECT COUNT(userID) INTO broj FROM dijeli_informacije
WHERE userID2=mojID;
SELECT * FROM korisnik_prikaz WHERE userID IN
(SELECT userID FROM dijeli_informacije WHERE userID2=mojID)
ORDER BY userID;
END//
delimiter ;
```

Broj korisnika koji s našim korisnikom dijeli informacije će biti u varijabli koju proslijedimo u `CALL` *iskazu kod pozivanja procedure. Na primjer, naredba*

```
CALL koliko_ljudi_sa_mnom_dijeli_informacije(1, @broj);
```

će u varijabli `broj` *pohraniti broj ljudi iz aktivne baze koji s korisnikom čiji je* `userID=1` *dijeli informacije. Naredba* `SELECT @broj;` *će ispisati taj broj.*

Klauzula `RETURNS` se može odrediti samo za funkciju i obvezna je. Određuje tip podatka koji funkcija vraća, a tijelo funkcije mora imati iskaz `RETURN <vrijednost>`. Ako `RETURN` iskaz vraća vrijednost drugačijeg tipa, vrijednost se pretvara u pravi tip.

Navedimo ovdje činjenice vezane uz `tijelo_procedure` i `tijelo_funkcije` (jednim imenom **tijelo rutine**) koje su vezane uz procedure i funkciju iz rada i koda, a više činjenica možemo naći u [12].

- Ukoliko je potrebno unutar rutine izvršiti više od jednog iskaza, osim ključnih riječi `BEGIN` i `END`, moramo postaviti drugu oznaku kao graničnik. (O tome smo već govorili u *potpoglavlju 3.1.2*).

- Forma pridruživanja je:

  ```
  SET <varijabla> = <izraz>;
  ```

  Vrijednost izraza s desne strane znaka jednakosti se izračuna ili nađe i potom pridruži vrijednosti varijable s lijeve strane znaka jednakosti. `NULL` je dozvoljen izraz. Izraz može biti čak i upit, toliko dugo dok vraća jednu vrijednost.

- Iskazi koji vraćaju skup rezultata se mogu koristiti unutar pohranjene procedure, ali ne unutar pohranjene funkcije. Ta zabrana uključuje `SELECT` iskaze koji nemaju `INTO <popis_varijabli>` klauzulu i druge iskaze kao npr. `SHOW` i `CHECK TABLE`.

Poslužitelj se nosi s tipom podatka parametra rutine, lokalne[48] varijable rutine stvorene naredbom `DECLARE <ime_varijable> <tip_varijable>` ili s vrijednošću koju funkcija vraća na sljedeći način:

- Zadaci se provjeravaju na nepodudaranje tipa podatka i preljev[49]. Problemi s pretvaranjem i preljevanjem podataka rezultiraju upozorenjem, ili greškom.

- Samo se skalarna vrijednost može pridružiti atributu.

Dobrom praksom pokazalo se izbjegavati davanje imena (procedurama ili funkcijama) za koje već postoji SQL funkcija.

3.3.2 Grananje

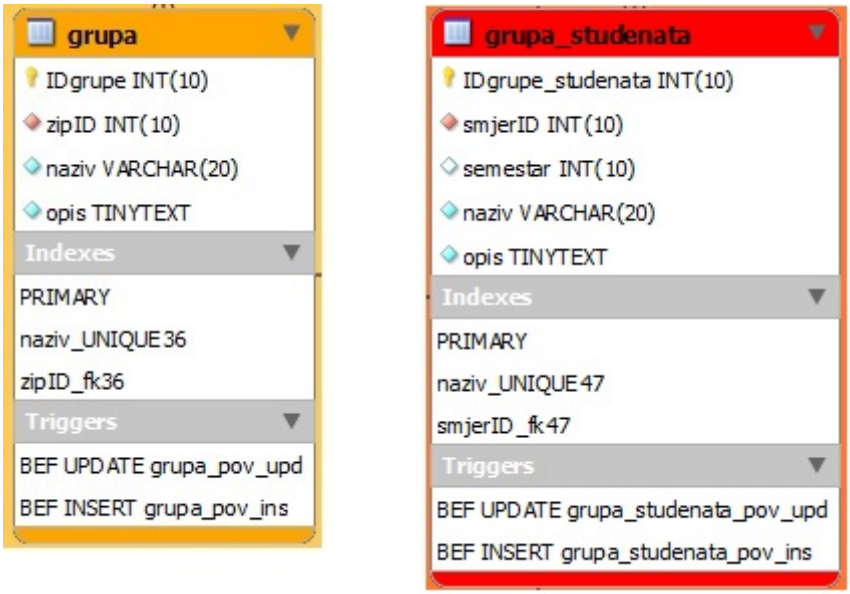
Promotrimo `IF` iskaz kad su u pitanju pohranjene procedure i funkcije. Općenita forma `IF` iskaza u MySQL-u je sljedeća:

```
delimiter <oznaka>
IF <uvjet> THEN
<popis iskaza>;
ELSEIF <uvjet> THEN
<popis iskaza>;
ELSEIF
...;
[ELSE
<popis iskaza>;]
END IF; <oznaka>
delimiter ;
```

[48] Radi se o varijablama čiju vrijednost SUBP ne čuva nakon izvođenja funkcije ili procedure. Deklaracije moraju prethoditi iskazima koji se izvršavaju u tijelu funkcije ili procedure.

[49] engl. overflow

Uvjet može biti bilo koja *boolean* vrijednost koja se može pojaviti u `WHERE` klauzuli SQL iskaza. Svaki **popis iskaza** se sastoji od iskaza koji završavaju znakom ';', ali ne treba biti okružen sa `BEGIN...END`. Zadnji `ELSE` i njegovi iskazi su opcionalni.



Slika 3.5 *Relacijske sheme relacija* `grupa` *i* `grupa_studenata`

Primjer 3.24 (Funkcija s grananjem) *Funkcija* `kojaGrupa` *prima broj (parametar* `broj`*) i string (parametar* `koja`*), te vraća konkatenaciju stringova za broj i specifičnost grupe ako grupa postoji, u suprotnom javlja da nema takve grupe. Zbog* `IF-THEN-ELSE` *naredbe je potrebna promjena graničnika prije i nakon funkcije* `(delimiter // ... delimiter ;)`*, a naredba* `CONCAT(string1, string2,...,stringN)` *vraća konkatenaciju stringova - u našem slučaju* `'Grupa'`*, broj (pretvoren u string) i ovisno o tome radi li se o grupi profesora ili studenata string* `'profesora'` *ili* `'studenata'`*. Donosimo sada kod funkcije.*

```
delimiter //
CREATE FUNCTION kojaGrupa (broj INT UNSIGNED, koja VARCHAR(20))
RETURNS VARCHAR(35)
IF koja='profesori' AND broj IN (SELECT IDgrupe FROM grupa)
THEN RETURN CONCAT ('Grupa ', broj, ' profesora.');
ELSEIF koja='studenti' AND broj IN
(SELECT IDgrupe_studenata FROM grupa_studenata)
THEN RETURN CONCAT ('Grupa ', broj, ' studenata.');
ELSE RETURN 'Nema takve grupe.';
END IF;//
```

```
delimiter ;
```

Funkciju bismo pozvali na sljedeći način i dobili bismo odgovore koji pišu kao komentar, pod pretpostavkom da imamo 11 grupa profesora i 10 grupa studenata:

```
SELECT kojaGrupa(1, 'nepostojeci'); /*'Nema takve grupe.'*/
SELECT kojaGrupa(1, 'studenti'); /*'Grupa 1 studenata.'*/
SELECT kojaGrupa(1, 'profesori'); /*'Grupa 1 profesora.'*/
SELECT kojaGrupa(17, 'studenti'); /*'Nema takve grupe.'*/
SELECT kojaGrupa(12, 'profesori'); /*'Nema takve grupe.'*/
```

Literatura

[1] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison-Wesley, SAD, 1995.

[2] R. Elmasri, S. Navathe, *Fundamentals of Database Systems, Sixth Edition*, Addison-Wesley, SAD, 2011.

[3] H. Garcia-Molina, J. D. Ullman, J. Widom, *Database Systems The Complete Book, Second Edition*, Pearson Prentice Hall, New Jersey, 2009.

[4] S. Jelić, *Baze podataka - radni materijal iz kolegija Dizajniranje i modeliranje baza podataka*, Odjel za matematiku, Sveučilište Josipa Jurja Strossmayera, Osijek, 2013.

[5] V. Mileusnić, *Modeliranje i dizajn baze podataka - završni rad*, Fakultet organizacije i informatike, Varaždin, 2011

[6] R. Ramakrishnan, J. Gehrke, *Database Management Systems, Third Edition*, McGraw-Hill Higher Education, New York, 2003.

[7] A. Silberschatz, H. F. Korth, S. Sudarshan, *Database System Concepts, Sixth Edition*, McGraw-Hill Connect Learn Succeed, New York, 2011. April 16-20, 2012, (2012), 839-848.

[8] `https://class.coursera.org/db/class/index`, 24.07.2013.

[9] `http://www.inpublic.hr/2012/11/infocus/kratka-povijest-drustvenih-mreza-1-dio/`, 08.01.2014.

[10] `http://www.inpublic.hr/2012/12/infocus/kratka-povijest-drustvenih-mreza-2-dio/`, 08.01.2014.

[11] *MySQL 5.6 Reference Manual*, Oracle, Redwood City, 2013.

[12] *MySQL 5.7 Reference Manual*, Oracle, Redwood City, 2013.

Sažetak

Sažetak. U ovom radu obrađena je baza podataka za sveučilišnu društvenu mrežu. Najprije je naveden logički dizajn baze podataka, pri čemu su postavljeni kriteriji za model baze, te je pokazano kako se baza nalazi u trećoj normalnoj formi. Potom se počinje govoriti o implementaciji baze podataka u MySQL-u, pri čemu je obrađena tematika ograničenja (ne `NULL` ograničenje, ograničenje na ključ i ograničenje na strani ključ) i okidača kao aktivnih elemenata SQL-a, nakon toga prikaza kao virtualnih relacija, indekasa kao strukture podataka koja nam pomaže brže pronaći odgovarajuće retke, te pohranjenih procedura i funkcija. Svaka od navedenih tema popraćena je primjerima i obrazloženjima primjera (vezanih uz implementaciju) koji se nalaze u kodu baze. U Dodacima se nalazi relacijska shema baze, te objašnjenje priloga predanih na CD-u uz rad.

Ključne riječi: baza podataka, društvena mreža, sveučilište

University social network database

Abstract. This paper discusses the topic of university social network database. First, the logic design for the database is given, wherein criteria are provided for the database model and it is shown that the database is in the third normal form. Next, the database implementation in MySQL is discussed, wherein constraints (not-NULL constraints, key constraints and foreign key constraints) and triggers as active elements in SQL are presented; and after that, the paper discusses views as virtual relations, indexes as data structures which help with faster search for tuples and stored procedures and functions. Each of the listed topics is followed by examples and explanations of examples (related to the database implementation) from the database code. In Additions we can find relational database schema and an explanation of accompanying material on CD.

Key words: database, social network, university

Životopis

Rođena sam 21. travnja 1989. u Koprivnici. U djetinjstvu sam zavoljela matematiku. Moj pokojni djed je bio profesor matematike, a kako sam dio djetinjstva provela kod bake i djeda, usvajala sam od djeda ljubav prema matematici. Od 1996. do 2004. godine pohađala sam *Osnovnu školu Frana Krste Frankopana* u Osijeku. U 4. razredu osnovne škole sam krenula na matematičare i kroz natjecanja sam sve više i više razvijala talent za matematiku. Od 4. razreda osnovne škole, sudjelovala sam na 20-ak natjecanja iz matematike. Dvaput sam sudjelovala na regionalnom natjecanju (u 5. i 6. razredu; u 5. razredu sam dobila pohvalu za 5. mjesto), a u 7. i 8. razredu osnovne škole sudjelovala sam na državnom natjecanju (u 7. razredu sam dobila pohvalu za 13.-14. rang). U 8. razredu sam sudjelovala i na županijskom natjecanju iz programiranja u *Q-Basic*-u, i tom prilikom osvojila sam drugo mjesto.

Nakon osnovne škole, upisala sam *III. gimnaziju, Osijek* (koju sam pohađala od 2004. do 2008. godine). U prvom razredu srednje škole dobila sam na državnom natjecanju iz matematike pohvalu za 12. rang. U drugom razredu srednje škole sam na županijskom natjecanju iz latinskog jezika osvojila 6. mjesto. U trećem i četvrtom razredu srednje škole, primala sam županijsku stipendiju za učenike.

Po završetku srednje škole, upisujem 2008. godine nastavnički studij na *Odjelu za matematiku*. Na trećoj godini studija primala sam stipendiju od Nacionalne Zaklade za potporu učeničkom i studentskom standardu.

Dodaci

U ovom poglavlju donosimo dodatke uz rad - relacijsku shemu baze i objašnjenje priloga koji se nalaze na CD-u priloženom uz rad.

A Relacijska shema baze

U ovom dodatku donosimo relacijske sheme svih relacija u bazi. Relacijske sheme navode se na način da se navede ime relacije, a potom popis njezinih atributa u zagradi. Primarni ključ svake relacije čine atributi koji su podcrtani, a strane ključeve čine atributi koji su ukošeni.

Relacijska shema baze za našu sveučilišnu društvenu mrežu je sljedeća:

- drzava(drzavaID, ime_drzave)
- naziv_grada(gradID, postanski_broj, naziv_grada, *drzavaID*)
- korisnik(userID, username, password, ime, prezime, spol, datum_rodjenja, *mjesto_rodjenja*)
- ulica(ulicaID, naziv_ulice, *gradID*)
- student(*sID*, *ulicaID*, broj, telefon, mobitel, email)
- nastavne_titule(ntitulaID, naziv_ntitule, kratica_ntitule)
- znanstvene_titule(ztitulaID, naziv_ztitule, kratica_ztitule)
- profesor(*profesorID*, *ntitulaID*, *ztitulaID*, *ulicaID*, broj, telefon, mobitel, email)
- aktivnost(*userID*, aktivnost)
- interes(*userID*, interes)
- glazba(*userID*, glazba)
- TV_emisije_i_serije(*userID*, TV_emisije_i_serije)
- filmovi(*userID*, filmovi)
- knjige(*userID*, knjige)
- o_sebi(*userID*, o_sebi)

- tekst_poruke(tekstID, tekst_poruke)
- poruke(*posiljateljID*, *primateljID*, vrijeme_slanja, *tekstID*)
- spomenar(IDzapisa, *tko_pise*, *kome_pise*, poruka, datum)
- ustanove_obrazovanja(ustanovaID, naziv_ustanove, *ulicaID*, broj, telefon, fax)
- obrazovanje_u_ustanovi(ID, *ustanovaID*, vrsta_obrazovanja)
- obrazovanje(IDobrazovanja, *ID*, godina_pocetka, godina_zavrsetka)
- obrazovanje_korisnika(*userID*, *IDobrazovanja*)
- slika(slikaID, *userID*, ime_slike, url_slike)
- o_slici(*slikaID*, o_slici)
- fakultet(*fakultetID*, web_stranica, *dekanID*)
- dodatno(*ustanovaID*, *ulicaID*, broj, telefon, fax)
- tip_studija(tipID, naziv_tipa)
- smjerovi_na_fakultetima(smjerID, naziv_smjera, *fakultetID*, *tipID*, traje_semestara)
- prof_na_fakultetima(*fakultetID*, *profesorID*, email, postotni_dio_radnog_vremena)
- kabineti(kabinetID, oznaka_kabineta, telefon, fax, *fakultetID*)
- kabineti_prof(*kabinetID*, *profesorID*)
- web_stranice_prof(*fakultetID*, *profesorID*, web_stranica)
- termini_konzultacija(terminID, danID, pocetak, kraj)
- konzultacije_prof(*fakultetID*, *profesorID*, *terminID*, datum_pocetka, datum_zavrsetka)
- znanstveno_istrazivacko_podrucje(zipID, ime_zip)
- zip_prof(*profesorID*, *zipID*)

- grupa(IDgrupe, *zipID*, naziv, opis)
- korisnici_grupe(*IDgrupe*, *profesorID*)
- post_grupe(postID, *IDgrupe*, *profesorID*, tekst, vrijeme_objave)
- komentari(IDkomentara, *postID*, *profesorID*, tekst, vrijeme_objave)
- status_studenta(statusID, naziv_statusa)
- studira_na(*sID*, *smjerID*, semestar, broj_indeksa, *statusID*)
- kolegij(kolegijID, naziv_kolegija)
- predznanja(*kolegijID*, *smjerID*, *potrebni_kolegiji*)
- izvodi_se(*kolegijID*, *smjerID*, web_stranica, semestar, ECTS)
- izvode(*kolegijID*, *smjerID*, *profesorID*)
- upisao(*sID*, *kolegijID*, *smjerID*)
- grupa_studenata(IDgrupe_studenata, *smjerID*, semestar, naziv, opis)
- korisnici_grupe_st(*IDgrupe_studenata*, *sID*)
- post_grupe_st(poststID, *IDgrupe_studenata*, *sID*, tekst, vrijeme_objave)
- komentari_st(IDstkomentara, *poststID*, *sID*, tekst, vrijeme_objave)
- dijeli_informacije(*userID*, *userID2*)
- jezici(jezikID, naziv_jezika)
- gostujuci_student(*gID*, *ustanovaID*)
- gost_zna_jezike(*gID*, *jezikID*)
- gosta_zanima(*gID*, *zipID*)
- gost_slusa(*gID*, *kolegijID*, *smjerID*)

B Prilozi

Ovdje ćemo navesti i ukratko pojasniti priloge uz ovaj diplomski rad, koji se nalaze na CD-u.

B.1 Grafički prikaz relacijske sheme baze `aktivna`

Grafički prikaz relacijske sheme baze `aktivna` nalazi se u 2 dokumenta:

- Kao veliki poster u PDF formatu - u dokumentu `grafickiPrikazPDF.pdf`,
- Kao MySQL Workbench model (u kojem možemo promatrati veze, pomicati tablice, promatrati ograničenja za strane ključeve itd.) - u dokumentu `grafickiPrikazMWB.mwb`.

Sam model baze je grupiran na 7 područja. Svako od prvih 6 područja sadrži one tablice koje su vezane uz njega, a sedmo područje čine prikazi (zelena boja). Navedimo sada tablice vezane uz svako od prvih 6 područja, te područje vezano uz prikaze:

Smeđe područje (*O korisniku*) sadrži tablice: `korisnik`, `aktivnost`, `interes`, `glazba`, `TV_emisije_i_serije`, `filmovi`, `knjige`, `o_sebi`, `slika`, `o_slici`, `spomenar`, `dijeli_informacije`, `poruke`, `tekst_poruke` i `obrazovanje_korisnika`.

Narančasto područje (*O profesorima*) sadrži tablice: `profesor`, `nastavne_titule`, `znanstvene_titule`, `prof_na_fakultetima`, `izvode`, `kabineti_prof`, `termini_konzultacija`, `konzultacije_prof`, `znanstveno_istrazivacko_podrucje`, `zip_prof`, `grupa`, `korisnici_grupe`, `post_grupe`, `komentari`, `web_stranice_prof`.

Crveno područje (*O studentima*) sadrži tablice: `student`, `status_studenta`, `studira_na`, `upisao`, `grupa_studenata`, `korisnici_grupe_st`, `post_grupe_st`, `komentari_st`.

Žuto područje (*Lokacije, ustanove i obrazovanje*) sadrži tablice `ustanove_obrazovanja`, `dodatno`, `obrazovanje_u_ustanovi`, `obrazovanje`, `ulica`, `naziv_grada`, `drzava`.

Ljubičasto područje (*O fakultetima*) sadrži tablice: `fakultet`, `tip_studija`, `smjerovi_na_fakultetima`, `kabineti`, `izvodi_se`, `kolegij`, `predznanja`.

Plavo područje (*O gostujućim studentima*) sadrži tablice: `gostujuci_student`, `jezici`, `gost_zna_jezike`, `gosta_zanima`, `gost_slusa`.

Zeleno područje (*Prikazi*) sadrži 82 prikaza.

B.2 Kod i objašnjenje koda aktivne baze

U ovom potpoglavlju kratko ćemo progovoriti o kodu aktivne baze i povijesne baze. Kod aktivne baze nalazi se u datoteci `aktivna.sql`, a kod povijesne baze u datoteci `povijesna.sql` na CD-u. Aktivna i povijesna baza imaju 57 tablica, 82 prikaza, 15 procedura i jednu funkciju (u tome im je kod jednak). Razlika u kodu je u tome što aktivna baza ima 128 okidača (za unos i ažuriranje podataka s povijesnom bazom, te za brisanje krivih podataka), a povijesna 14 (samo za brisanje krivih podataka). Pri početku je iskomentiran kod koji briše funkciju, procedure, prikaze, okidače i tablice, i to redoslijedom obrnutim od onoga kojim su nastajali. Taj je dio koda bio koristan pri fragmentalnom izvršavanju koda, a smatrali smo bitnim ostaviti ga (pod komentarom) radi mogućnosti testiranja baza. Objasnimo sada pojedine dijelove koda na primjeru aktivne baze.

B.2.1 Tablice

Svaka od 57 tablica nastala je pomoću ključnih riječi `CREATE TABLE IF NOT EXISTS`, nakon čega je navedeno ime tablice, kao i popis atributa, a potom slijedi ograničenje na ključ (odnosno određivanje primarnog ključa za svaku tablicu) i ograničenje stranog ključa (odnosno uvjet referencijalnog integriteta) ako postoji atribut ili skup atributa koji čini strani ključ u toj tablici. Svaka je tablica određena kao `InnoDB` tablica, kako bi strani ključevi doista zadovoljavali uvjet referencijalnog integriteta.

B.2.2 Indeksi

Odmah nakon kreiranja pojedine tablice slijedi kreiranje indeksa na tu tablicu (ako je indeks potreban). Kreirali smo indekse za atribute ili skupove atributa za koje smo procijenili da bi upiti u bazi bili česti. Za primarni ključ svake tablice nismo posebno stvarali indeks, jer ga MySQL sam stvara. Uz svaki indeks piše i komentar.

B.2.3 Okidači

Okidači su u bazi najčešće korišteni za:

- unos podataka iz aktivne u povijesnu bazu (svaki takav okidač imao bi ime `imeTablice_pov_ins`, a unosio bi podatke u povijesnu bazu prije unosa u aktivnu bazu `BEFORE INSERT ON aktivna.imeTablice`),
- ažuriranje podataka iz aktivne baze s povijesnom bazom (svaki takav okidač bi imao ime `imeTablice_pov_upd`, a ažurirao bi podatke najprije u povijesnoj bazi, a potom u aktivnoj bazi - `BEFORE UPDATE ON aktivna.imeTablice`, pri čemu bi redak koji treba

ažurirati prepoznao po primarnom ključu - primjer za prepoznavanje ključa koji bi se sastojao od jednog atributa je `WHERE atribut=OLD.atribut`),

- ulančano brisanje podataka u aktivnoj i povijesnoj bazi koji ne bi odgovarali pojedinim tablicama jer ne zadovoljavaju uvjet (npr. termin konzultacija kada konzultacije završe prije nego počnu) - najprije se na unos u tablicu aktivne baze aktivira okidač u aktivnoj bazi koji iz istoimene tablice povijesne baze briše takve retke, a potom se na brisanje redaka iz te tablice povijesne baze aktivira okidač koji iz istoimene tablice aktivne baze briše takve retke.

B.2.4 Prikazi

Prikazi se u bazi najviše koriste za tablice koje imaju strane ključeve, kako bi podaci bili potpunije prikazani. Ukoliko tablica ima više stranih ključeva, te je više spojeva potrebno napraviti kako bi prikaz bio potpuniji, korišteni su pomoćni prikazi (pogledati komentare uz prikaze u kodu). Ukoliko tablica `imeTablice` ima prikaz, on je nazvan `imeTablice_prikaz`.

Osim navedenih prikaza, u bazi postoji još 7 specifičnih prikaza:

1. Prikaz `osobni_podaci` nam donosi aktivnost, interes, glazbu, TV emisije i serije, filmove, knjige koje korisnik voli, kao i ono što bi sam rekao o sebi - sve na jednom mjestu.

2. Prikaz `spomenar_obrisani` nam donosi popis zapisa u spomenarima koji potječu od osoba koje su obrisane.

3. Prikaz `ustanove_i_adrese` nam donosi popis svih lokacija svih ustanova, što je korisno zbog nekih ustanova koje se nalaze na više lokacija.

4. Prikaz `termini_konzultacija_prikaz` nam pomoću `SELECT CASE` klauzule umjesto atributa `danID` stavlja ime dana kad se održavaju konzultacije.

5. Prikaz `tko_moze_u_grupu_prof` nam pokazuje koji profesori mogu u koju grupu profesora (jer se bave tim znanstveno-istraživačkim područjem za koje je grupa namijenjena).

6. Prikaz `kolegijsmjerfakultet` nam pokazuje koji se kolegiji izvode na kojem fakultetu.

7. Prikaz `kolegijsmjerprofesor` nam pokazuje koji bi kolegij na kojem smjeru mogao koji profesor izvoditi (jer se izvodi na fakultetu na kojem se taj profesor radi).

B.2.5 Procedure

U bazi je korišteno 15 procedura. Ovdje navodimo njihov popis uz kratku naznaku čemu služi, a u kodu je pod komentarom uz svaku proceduru detaljnije objašnjeno što i kako svaka procedura radi:

1. Procedura `koga_vidim` pokazuje osobne podatke onih korisnika koji su ih s nama podijelili.
2. Procedura `koje_poslane_poruke_vidim` pokazuje naše poslane poruke.
3. Procedura `koje_primljene_poruke_vidim` pokazuje naše primljene poruke.
4. Procedura `kome_vidim_spomenar` pokazuje zapise u spomenarima onih osoba koje su s nama podijelile informacije.
5. Procedura `koje_postove_vidim_u_grupama` pokazuje koje postove vidimo u grupama.
6. Procedura `koje_postove_vidim_u_kojoj_grupi` pokazuje koje postove vidimo u kojoj grupi.
7. Procedura `koje_komentare_vidim` pokazuje koje komentare vidimo.
8. Procedura `koje_komentare_vidim_u_kojoj_grupi` pokazuje koje komentare vidimo u kojoj grupi.
9. Procedura `koliko_ljudi_sa_mnom_dijeli_informacije` nam pokazuje koliko korisnika je s nama podijelilo informacije i koji su to korisnici.
10. Procedura `s_koliko_ljudi_dijelim_informacije` nam pokazuje s koliko korisnika smo podijelili informacije i koji su to korisnici.
11. Procedura `razmjena_poruka` nam pokazuje razmjenu poruka između dva navedena korisnika.
12. Procedura `nadji_drzavaID` nam daje atribut `drzavaID` na temelju unešenog naziva države.
13. Procedura `nadji_grad` nam ispisuje podatke o gradu na temelju unešenog naziva grada.
14. Procedura `pretrazivanje_po_imenu_i_prezimenu` nam ispisuje podatke o korisniku iz prikaza korisnika kojima su ime i prezime jednaki onima koje smo unijeli.
15. Procedura `student_upisuje_kolegij` nam na temelju unešenog korisničkog imena studenta, imena kolegija i atributa `smjerID` upisuje studenta na kolegij na tom smjeru.

B.2.6 Funkcija

U radu imamo jednu funkciju, nazvanu `kojaGrupa`. Kod funkcije i objašnjenje se već nalaze u *Primjeru 3.24*.